TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2612 – Elektrotechnika a informatika
Studijní obor: 1802R022 – Informatika a logistika

Tvorba softwaru, který vyhodnocuje nejkratší cestu na vytvořené městské pouliční síti

Creating software which evaluates the shortest path on the existing municipal street network

Bakalářská práce

Autor: Tomáš Chaloupek
Vedoucí práce: Ing. Josef Chudoba, Ph.D.

V Liberci 8. 9. 2013

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že Technická univerzita v Liberci má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

Abstrakt

Tato bakalářská práce se zabývá tvorbou softwaru vyhledávajícího nejkratší cestu mezi dvěma body v pouličním modelu města. Aplikace je napsána pomocí programovacího jazyka C#, který spadá pod .NET. Městský model bude representován datovým souborem, se kterým bude aplikace poté pracovat. V této práci datový soubor obsahuje město Liberec. Vstupní body do aplikace jsou omezeny pouze na křižovatky, není tedy možné zadat si vyhledávání nejkratší cesty mezi dvěma body na ulici. V oblasti ovládání aplikace umožňuje vyhledávat nejkratší cestu jen pro určité uživatelem nastavené kategorie pozemních komunikací např. jen přes silnice 1. třídy. Program dále umožňuje exportovat výslednou cestu do textového souboru. Aplikace má pouze textové uživatelské rozhraní, což znamená, že vstupní body jsou zadávány pomocí názvů křižovatek a výstup je dán seznamem těchto názvů.

Abstract

This work deals with the creation of a software which searches the shortest path between two points in a city streets model. The application is written using the C# programming language, which is a part of .NET. The urban model will be represented by a data file, with which the application will then work. In this work, the data file contains the city of Liberec. Entry points into the application are limited to intersections, it is not possible to specify a search of the shortest path between two points in a street. This application allows to search the shortest path only for specific user-defined categories of roads. The program also allows to export the resulting path into a text file. The application has a text-only user interface, which means that input points are entered by names of intersections and output is given by a list of these names.

Obsah

Seznam obrázků

1 Úvod

Cílem bakalářské práce je tvorba softwaru vyhodnocujícího nejkratší cestu mezi dvěma body v dané městské pouliční síti. Dalším úkolem této práce je vytvořit model pouliční sítě konkrétního města, v tomto případě jde o Liberec. Model města bude representován datovým souborem, se kterým bude aplikace poté pracovat. Datový soubor přiřazuje skutečnou kategorii pozemní komunikace pro každou cestu mezi dvojicí křižovatek. Tento software je napsán pomocí frameworku WinForms, který spadá pod soubor programátorských technologií .NET. Jako programovací jazyk je zde použit jazyk C#.

Text práce se nejprve zabývá teoretickým pojednáním o tom, jakými způsoby se řeší problematika vyhledávání nejkratší cesty. Vyhledávací algoritmus je popsán v kapitole 2.3. Způsobem, jakým se tvoří datový soubor obsahující městskou síť, se zabývá kapitola 3. Další část textu obsažená v kapitole 5 se věnuje popisu zdrojového kódu aplikace. Kapitola 6 se věnuje popisu uživatelského rozhraní tedy ovládání. Konec je věnován srovnání tohoto software s profesionálními aplikacemi jako jsou Mapy.cz nebo Google maps, které lze nalézt v kapitole 7.

2 Teorie

Nejprve je potřeba vytvořit model daného města. Nejlepší způsob, jak problému tvorby modelu přistupovat je, nahlížet na město pomocí teorie grafů. To znamená, že vytvoříme graf, který bude reprezentovat konkrétní město. Struktura tohoto grafu bude popsána v datovém souboru popisujícím dané město. Způsob tvorby datového souboru je popsán v kapitole 3.

Teorie grafů definuje množství grafových algoritmů, které můžeme použít pro zkoumání grafů. Pro řešení problému vyhledávání nejkratší cesty se využívá Dijkstrův a Floyd-Warshallův algoritmus. V tomto programu bude využit algoritmus Dijkstrův, který vyhledává nejkratší cestu mezi dvojicí zadaných vrcholů.

2.1 Náhled na silniční síť jako na graf

Silniční síť si můžeme znázornit grafem. Křižovatky odpovídají vrcholům grafu a silnice mezi nimi hranám. Každý úsek silnice odpovídající hraně má svojí délku. Ta odpovídá ohodnocení hrany. Otázkou tedy je, jaká je nejkratší cesta mezi určitou dvojicí křižovatek a jak je tato cesta dlouhá. Maximální povolená rychlost na každém úseku silnice nám říká, jak rychle po ní můžeme jet. Z rychlosti a délky silnice se dá dopočítat, za jak dlouhou úsek silnice projedeme. Proto se můžeme ptát i na nejrychlejší cestu mezi zvolenými křižovatkami. Nejrychlejší cesta nemusí být zrovna nejkratší (Černý, 2010). Náš program ovšem nebude řešit problém nejrychlejší cesty, ale pouze nejkratší cestu.

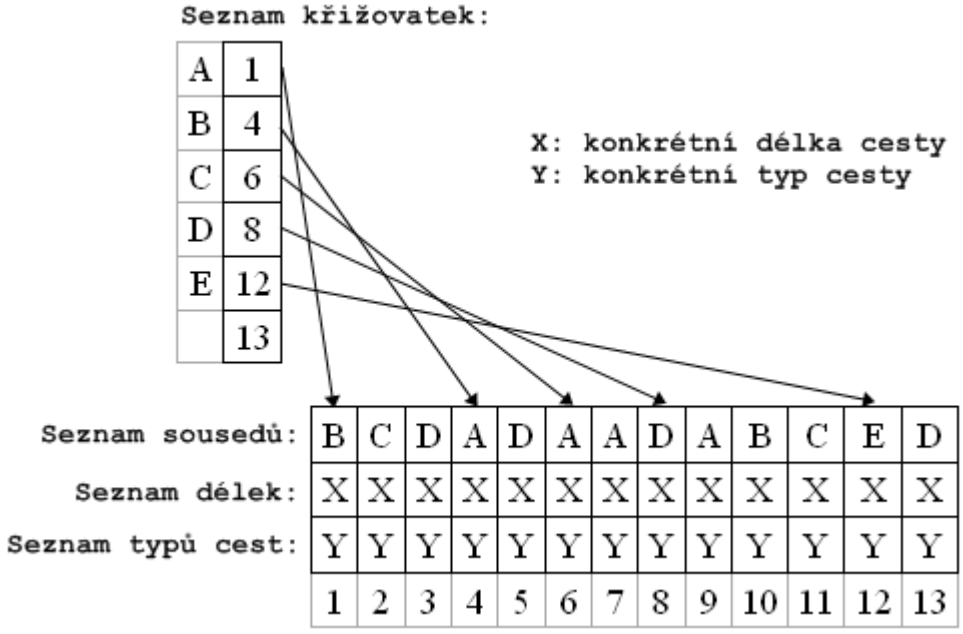
Ve městech kromě běžných obousměrných silnic existují i jednosměrné komunikace. Graf takovýchto silničních sítí je representován orientovaným grafem. Je tedy zcela přirozené hledat nejkratší cestu i v orientovaných grafech. Vzdálenost dvou vrcholů s a t v orientovaném grafu je opět délka nejkratší orientované cesty z s do t. Všechny algoritmy, které si v této kapitole uvedeme, fungují i pro orientované grafy. To je z důvodu, že i neorientovaný graf, ve kterém hledáme cestu, máme reprezentovaný jako orientovaný graf. Každou hranu si pamatujeme jako dvě orientované hrany (šipky) vedoucí proti sobě (Černý, 2010).

Dále popisované algoritmy fungují pouze pro kladné ohodnocení hran. Nezápornost hran je zde zajištěna, protože se jedná o vzdálenosti a ty nemohou být záporné. Pokud by

ohodnocení hran bylo záporné, nebylo by možné zaručit správnost algoritmu. Pro grafy se záporným ohodnocením hran se používá Bellman-Fordův algoritmus (Černý, 2010).

2.2 Reprezentace grafu v počítači

Městská síť je representována grafem. Ten je v počítači representován jako seznam sousedů. Význam seznamu sousedů je vidět na obrázku 1.



Obrázek 1 - Graf jako seznam sousedů

Vrcholy jsou chápány jako indexy pole, na jehož položkách jsou uloženy různé informace o vrcholu. Tyto informace mohou být vzdálenost od startovního vrcholu, předcházející vrchol. V případě seznamu sousedů máme dvě pole. První pole je *Seznam křižovatek* a druhé *Seznam sousedů*. *Seznam křižovatek* určuje pro každý vrchol, na jakém indexu začínají sousední vrcholy v poli *Seznam sousedů*. Rozměr pole *Seznam křižovatek* musí být o jedničku větší, než je počet vrcholů. To je z důvodu, aby bylo možné zjistit, kde končí sousední vrcholy posledního vrcholu v poli *Seznam křižovatek*. Pole *Seznam sousedů* tedy slouží k ukládání sousedních vrcholů pro každý vrchol. Souběžně s tímto polem zde může být definováno více polí. To se hodí v případě, kdy je třeba ukládat o vrcholech více informací, než jen jaké jsou sousední vrcholy každého vrcholu. Například v této aplikaci je třeba ukládat ještě vzdálenost sousedních vrcholů a typ cest k sousedním

vrcholům. Deklarujeme tedy další dvě pole *Seznam délek* a *Seznam typů cest*, které mají stejný rozměr jako pole *Seznam sousedů* a budeme do nich ukládat vzdálenosti a typy cest. Při procházení sousedních vrcholů je poté možné sledovat, jak jsou vzdálené a jaký typ cesty k nim vede.

2.3 Dijkstrův algoritmus

Dijkstrův algoritmus najde nejkratší cestu v orientovaném grafu s nezáporným ohodnocením hran (Černý, 2010). Jeho funkce je taková, že vyhledává nejkratší cesty od zvoleného startovního vrcholu *s* ke všem ostatním vrcholům. Lze jej tedy použít pro tyto případy:

1. vyhledáváme nejkratší cesty od vrcholu *s* ke všem ostatním vrcholům.
2. hledáme nejkratší cestu mezi vrcholem *s* a vybraným vrcholem *t*.

V této práci budeme k problému přistupovat pomocí bodu 2, protože vyhledáváme cestu mezi startovním a cílovým vrcholem.

Dijkstrův algoritmus využívá dvě množiny pro práci s vrcholy. První je množina trvalých vrcholů, kam se ukládají vrcholy, u kterých je už jasné, kolik měří nejkratší cesta mezi nimi a startovním vrcholem. Ve skutečnosti tato množina není algoritmem využita, protože není důležitá. Aktuální nejkratší vzdálenosti se ukládají do pole vzdáleností, takže tyto informace máme zde. Název množina trvalých vrcholů je zde použit proto, aby byl definován význam vrcholu, který je trvalý. U takového vrcholu se již nebude měnit hodnota nejkratší cesty.

Druhou množinou algoritmu je množina dočasných vrcholů. Ta je representována prioritní frontou, která představuje důležitou datovou strukturu algoritmu. V této frontě jsou umístěny vrcholy, jejichž délka nejkratší cesty mezi nimi a startovním vrcholem se může v průběhu činnosti algoritmu měnit. Priorita ve frontě má význam vzdálenosti mezi daným a startovním vrcholem. Do prioritní fronty do prioritní části vrcholů tedy ukládáme aktuální hodnoty těchto nejkratších vzdáleností.

2.3.1 Inicializace datových struktur algoritmu

Nejprve je třeba před spuštěním Dijkstrova algoritmu inicializovat jeho datové struktury. To znamená nastavit je na určitou výchozí hodnotu. Množina trvalých vrcholů je před startem algoritmu samozřejmě prázdná, ale vzhledem k jejímu nevyužití žádnou strukturu nenastavujeme. Struktury, které jsou využity, jsou tyto:

1. *pole vzdáleností* – zde jsou uloženy minimální vzdálenosti mezi daným vrcholem a startovním vrcholem. Daný vrchol je representován indexem. Každou položku pole je nutno nastavit na maximální možnou hodnotu. Jak velká tato hodnota to bude, závisí na zvoleném typu pole. Maximální hodnoty nastavujeme u všech položek pole kromě položky na indexu startovního vrcholu, tu nastavíme na nulovou hodnotu. Tak je to z toho důvodu, že je jasné, že vzdálenost od startovního vrcholu ke startovnímu vrcholu je nulová.
2. *pole předků* – zde je uložen strom nejkratší cesty. Položky tohoto pole je nutno vynulovat. Vynulování je nutné z důvodu, abychom měli jistotu, že startovní vrchol nebude mít žádného předka. V opačném případě by rekonstrukce výsledné cesty nemusela odpovídat skutečnosti.
3. *prioritní fronta* – tuto frontu je třeba naplnit vrcholy s danými prioritami. Výchozí priority máme uloženy v poli vzdáleností. Vzhledem k tomu, že vrcholy jsou representovány jako indexy, tak je možné do fronty vložit dvojici index a vzdálenost na daném indexu v poli vzdáleností. Index představuje vrchol a vzdálenost jeho prioritu.

2.3.2 Funkce algoritmu

Pseudokód algoritmu tak jak je použit v aplikaci, vypadá takto:

```
Inicializace datových struktur algoritmu
do{
 u = fronta.Dequeue();
 List sousediU = new List();
```

```
 List vzdalenostiSouseduU = new List();
 for(int i=SeznamKrizovatek[u]; i<SeznamKrizovatek[u+1]; i++){
  if (FiltrujSousedniVrchol(i)){
   sousediU.Add(SeznamSousedu[i]);
   vzdalenostiSouseduU.Add(SeznamDelek[i]);
  }
 }
 for (int i = 0; i < sousediU.Count; i++){
  vzdalenost = poleVzdalenosti[u] + vzdalenostiSouseduU[i];
  if (vzdalenost < poleVzdalenosti[sousediU[i]]){
   fronta.DecreaseKey(vzdalenost, sousediU[i]);
   poleVzdalenosti[sousediU[i]] = vzdalenost;
   polePredku[sousediU[i]] = u;
  }
 }
} while (fronta.Delka != 0);
```

Algoritmus je tvořen cyklem *do-while*, ve kterém je jeho funkčnost. Tento cyklus ukončí svou činnost ve chvíli, kdy přestane platit podmínka, že délka fronty je nenulová. To znamená, že algoritmus je ukončen, pokud délka fronty je rovna hodnotě nula a tedy neobsahuje již žádný vrchol.
Posloupnost kroků algoritmu je tato:

1. výběr vrcholu z prioritní fronty.
2. nalezení vyhovujících sousedních vrcholů vrcholu z bodu 1.
3. výpočet aktuální vzdálenosti mezi i-tým sousedním vrcholem a startovním vrcholem.
4. porovnání vypočtené a stávající vzdálenosti. Na základě výsledku porovnání aktualizujeme pole vzdáleností, pole předků a prioritní frontu.

Bod 1 obsahuje spuštění metody *Dequeue*, která je metodou prioritní fronty. Tato metoda vyjme vrchol s nejmenší prioritou z fronty a uloží ho do proměnné *u*. Dále je zde vytvoření seznamů pro ukládání sousedních vrcholů a vzdáleností k nim. Sousední vrcholy se ukládají do seznamu *sousediU* a vzdálenosti k sousedům do seznamu *vzdalenostisouseduU*.

Bod 2 obsahuje *for* cyklus, jehož řídící proměnná *i* je omezena tak aby prošla pouze sousední vrcholy vrcholu *u*. Tím pádem cyklus projde všechny sousední vrcholy vrcholu *u* a pro každý spustíme metodu *FiltrujSousedniVrchol*. Tato metoda na základě daných vlastností vrcholu určí, jestli se jedná o vhodného souseda nebo ne. Pokud se nejedná o vhodný vrchol, pak nebude přidán do seznamů *sousediU* a *vzdalenostiSouseduU* a nebude přes něj vyhledávána nejkratší cesta. V opačném případě bude vrchol přidán do těchto seznamů a bude se tak přes něj vyhledávat nejkratší cesta.

Bod 3 způsobí výpočet aktuální nejkratší vzdálenosti mezi daným sousedem a startovním vrcholem. Tato vzdálenost se spočítá jako součet dvou vzdáleností. První vzdálenost je mezi startovním vrcholem a vrcholem *u*, ta je již definitivní, protože vrchol *u* je označen jako trvalý (není v prioritní frontě). Druhá vzdálenost je délka hrany k danému sousednímu vrcholu vrcholu *u*. Tyto dvě vzdálenosti sečteme a budeme s nimi pracovat v bodě 4.

Bod 4 porovnává aktuálně vypočítanou vzdálenost z bodu 3 s dosavadní vzdáleností tohoto souseda od startovního vrcholu. Pokud platí, že nově vypočítaná vzdálenost je menší než dosavadní, znamená to, že jsme do tohoto sousedního vrcholu našli kratší cestu přes vrchol *u*. Proto je třeba tuto informaci zapsat do datových struktur algoritmu. Tento zápis provedeme pomocí těchto tří kroků:

1. do pole předků na pozici aktuálně řešeného souseda uložíme vrchol *u*.
2. do pole vzdáleností uložíme na stejnou pozici vzdálenost vypočítanou v bodě 3.
3. snížíme prioritu tohoto vrcholu v prioritní frontě. To uděláme pomocí metody *DecreaseKey*. Do této metody je třeba vložit nově vypočítanou vzdálenost z bodu 3 a daný sousední vrchol.

Pokud podmínka, že vzdálenost z bodu 3 není menší než dosavadní vzdálenost, splněna není, pak jsme nenašli kratší cestu, a proto není třeba nic zaznamenávat.

Po projití všech sousedních vrcholů vrcholu *u* následuje výběr dalšího vrcholu z prioritní fronty. Pro tento nový vrchol a jeho sousedy se bude opět opakovat posloupnost kroků 1 – 4.

Takovýto postup probíhá až do doby, kdy se vyprázdní prioritní fronta a všechny vrcholy se tak stanou trvalými. Poté je u všech vrcholů jasné, kolik měří nejkratší cesta mezi startovním vrcholem a kterýmkoliv jiným vrcholem. Pomocí pole předků jsme schopni si zrekonstruovat nejkratší cestu a v poli vzdáleností na indexu cílového vrcholu nalezneme, jak je nejkratší cesta dlouhá.

Algoritmus je konečný, protože v každé iteraci prohlásí jeden vrchol za trvalý. Vrcholů je jen *n* a z každého vede nejvýše *n-1* hran. Podobně pokud hledáme pouze nejkratší cestu do vrcholu *t* a ne do všech vrcholů, tak můžeme skončit v momentě, kdy bude *t* prohlášen za trvalý (Černý, 2010).

Výsledek algoritmu je uložen v poli předků. Toto pole si můžeme představit tak, že každý vrchol *v* má v poli předků uloženého svého předchůdce na nejkratší cestě do startovního vrcholu *s*. Pole předků je tedy směrovka, která říká, kudy máme vrchol *v* opustit, abychom se dostali zpět do vrcholu *s* po nejkratší cestě. Je-li je graf souvislý, tak z každého vrcholu *v* existuje cesta do *s* a každý vrchol kromě *s* má právě jednoho předchůdce na nejkratší cestě. Hrany {*v*, *polePredku[v]*} tvoří strom orientovaný do kořene startovního vrcholu *s*, který nazveme strom nejkratší cesty. Cílem hledání nejkratší cesty v grafu je tedy najít strom nejkratší cesty (Černý, 2010).

Nejkratší cestu mezi zadanými vrcholy *s* a *t* je možno najít pomocí rekurzivní metody. To provedeme tak, že si v poli předků vyhledáme cílový vrchol *t*. Na této položce pole nalezneme jeho předka, který má opět svého předka. Takto naše pole předků projdeme až ke startovnímu vrcholu s, který již žádného předka nemá.

Dijkstrův algoritmus je také popsán v jiných knihách viz Mareš, Nešetřil, Matoušek.

2.4 Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus se používá v případech, kdy je třeba zjistit nejkratší cesty mezi všemi dvojicemi vrcholů. V naší práci si ovšem vystačíme s Dijkstrovým algoritmem, protože hledáme nejkratší cestu pouze mezi jednou dvojicí křižovatek. Stejně jako u předchozího algoritmu je i zde vyžadována podmínka nezáporného ohodnocení hran (Černý, 2010).

Pokud chceme spočítat vzdálenost každé dvojice vrcholů, tak můžeme n-krát použít Dijkstrův algoritmus (na každý vrchol). Lepší možností je použít Floyd-Warshallův algoritmus, který počítá všechny vzdálenosti přímo, proběhne rychleji než n-krát použitý Dijkstrův algoritmus a ještě se snadněji implementuje. Implementace je tak jednoduchá, že pokud nám nebude záležet na časové složitosti, ale jen na rychlosti naprogramování, tak je lepší volbou než Dijkstrův algoritmus (Černý, 2010).

Vrcholy grafu očíslujeme čísly od jedničky do n. Vzdálenosti mezi každou dvojicí vrcholů si budeme ukládat do matice $n \times n$. Celý trik Floyd-Warshallova algoritmu spočívá v tom, že vzdálenosti nepočítáme přímo, ale v n iteracích (Černý, 2010).

V i-té iteraci spočítáme matici D^i. Hodnota $D^i[u; v]$ je délka nejkratší cesty z u do v, která smí procházet pouze přes vrcholy $\{1, 2, \ldots, i\}$. Jinými slovy $D^i[u; v]$ je délka nejkratší cesty v podgrafu indukovaném vrcholy $\{1, 2, \ldots, i\}$. V nulté iteraci začneme s maticí D^0. Hodnota $D^0[u; v]$ je délka hrany uv, pokud z u vede hrana do v, nula na diagonále a nekonečno jinak. Matice D^0 je tedy matice vzdáleností. V poslední iteraci skončíme s maticí D^n, která už bude obsahovat hledané vzdálenosti, protože cesty mezi u a v smí procházet přes všechny vrcholy (Černý, 2010).

Nejkratší cesta mezi u a v, která smí procházet pouze přes vrcholy $\{1, 2, \ldots, i\}$, buď projde přes vrchol i a nebo ne. Pokud nejkratší cesta neobsahuje i, tak je její délka $D^{i-1}[u; v]$. V opačném případě cestu rozložíme na dva úseky – před příchodem do i a po jeho opuštění. Ani jeden z úseků neobsahuje i a tak je délka této cesty $D^{i-1}[u; i] + D^{i-1}[i; v]$ (Černý, 2010).

K výpočtu $D^i[u; v]$ potřebujeme znát jen hodnoty $D^{i-1}[u; v]$, $D^{i-1}[u; i]$, $D^{i-1}[i; v]$, ale poslední dvě hodnoty se během i-té iterace nezmění. Proto můžeme nové hodnoty $D^i[u; v]$ zapisovat do stejné matice jako předchozí iteraci. Přepsanou položku $D^{i-1}[u; v]$ už nebude během iterace potřebovat. V celém algoritmu si tedy vystačíme jen s jednou maticí $D[*; *]$, do které budeme zapisovat všechny iterace (Černý, 2010).

3 Popis datového souboru

Datový soubor, který je representován textovým souborem, určuje městskou pouliční síť. Filosofie tvorby křižovatek je taková, že na každém řádku je popsána jedna křižovatka. Její struktura je takováto:

```
ID křižovatky+jméno křižovatky*definice 1. souseda|…|definice
n. souseda (1)
```

Křižovatka může mít libovolné množství sousedů. U těch je třeba definovat trojici údajů – jméno sousední křižovatky, vzdálenost k sousední křižovatce a typ cesty k sousední křižovatce. Do souboru se definice sousedů zadávají takto:

```
Jméno souseda,vzdálenost k sousedovi,typ cesty k sousedovi (2)
```

Tato definice sousedů (*2*) se dosadí do definice křižovatky (*1*). V obou definicích se nacházejí oddělovací znaky. Jde o tyto znaky: + * | „. Oddělovací znaky jsou důležité pro správnou konstrukci grafu, který representuje síť křižovatek. Není možné je zaměnit za jiné nebo snad odstranit. Tímto způsobem vznikají křižovatky tvořící celou městskou síť. Názvy křižovatek jsou dány ulicemi, které se v nich kříží a jsou odděleny podtržítkem. Obecný příklad vzniku jména křižovatky vypadá následujícím způsobem:

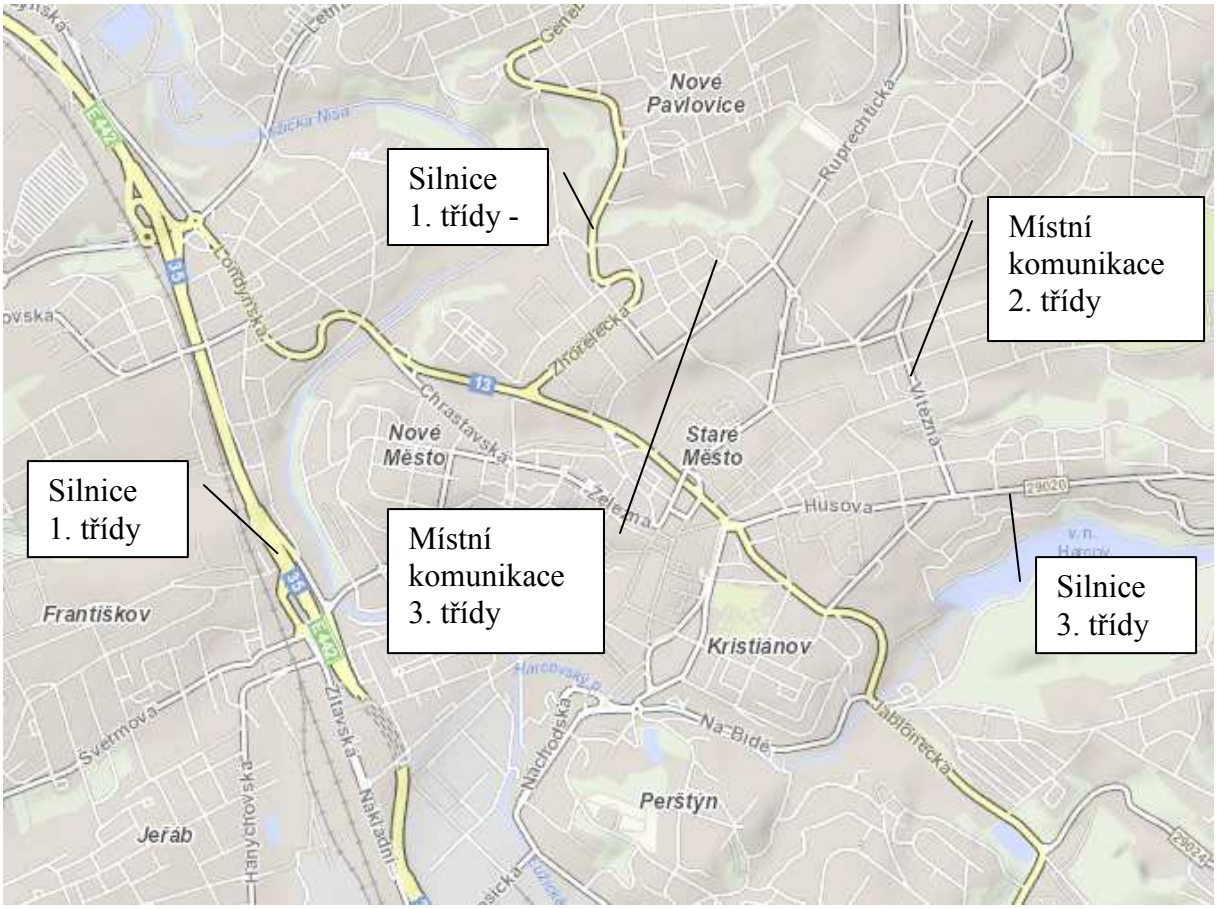
```
Název 1. ulice_název 2. ulice_..._název n. ulice (3)
```

Typ cesty je určen podle skutečné kategorizace ulic. Jedná se o tyto kategorie pozemních komunikací:

1. `Silnice 1. třídy` – žlutá a číselné označení na modrém podkladu.
2. `Silnice 2. třídy` – žlutá a číselné označení na žlutém podkladu.
3. `Silnice 3. třídy` – bílá a číselné označení na bílém podkladu.
4. `Místní komunikace 1. třídy` – v Liberci nejsou.

5. `Místní komunikace 2. třídy` – bílé s tučnými okraji.
6. `Místní komunikace 3. třídy` – bílé se slabými okraji.

Na následujícím obrázku vidíme mapu silniční sítě s rozlišenými kategoriemi pozemních komunikací.



Obrázek 2 - Značení kategorií pozemních komunikací na mapě. Zdroj: http://mapa.dopravniinfo.cz

V datovém souboru a ve zdrojovém kódu vlastní aplikace jsou tyto kategorie určeny výčtovým typem *TypCesty*, který obsahuje tyto hodnoty:

1. `nic = -1` žádná ulice
2. `chodci = 0` cesta pro pěší
3. `mistniKomunikace1 = 1` místní komunikace 1. třídy
4. `mistniKomunikace2 = 2` místní komunikace 2. třídy
5. `mistniKomunikace3 = 3` místní komunikace 3. třídy
6. `silnice2 = 4` silnice 2. třídy

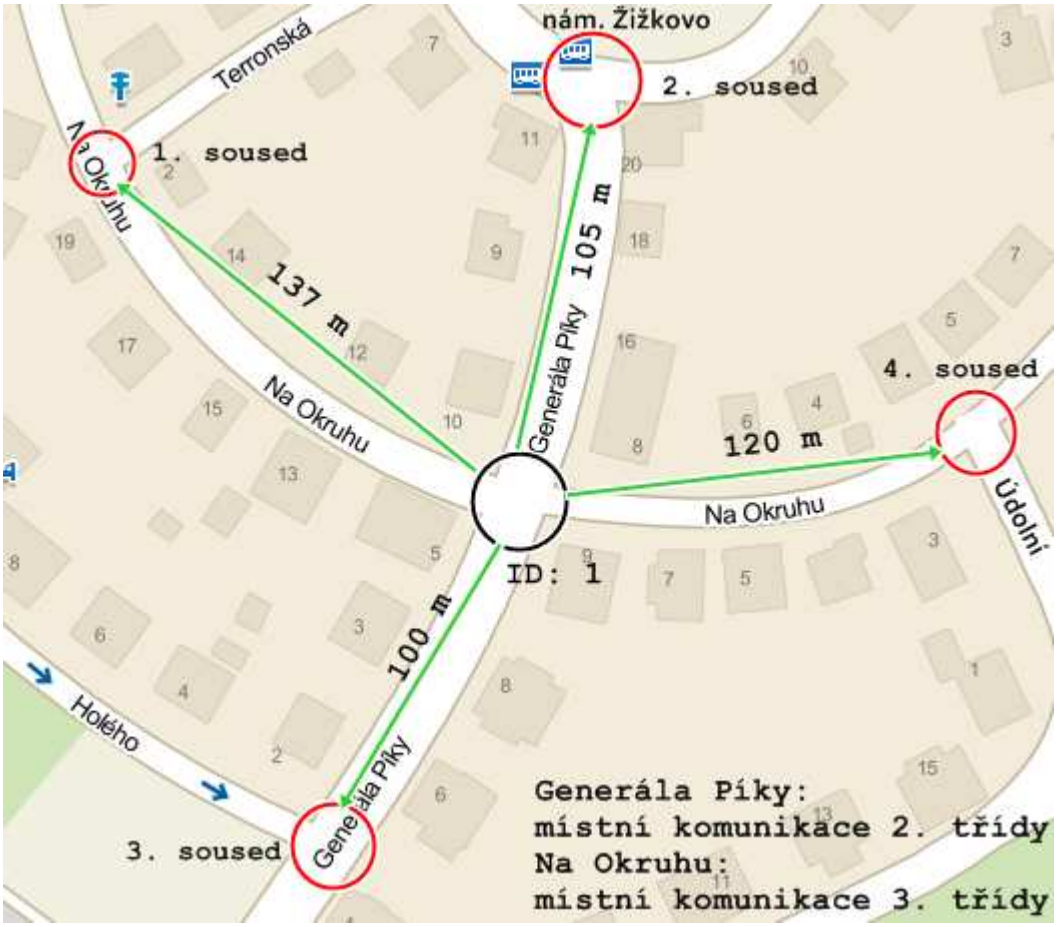
7. `silnice3 = 5` silnice 3. třídy
8. `silnice1AutaChodci = 6` silnice 1. třídy pro auta i chodce
9. `silnice1Auta = 7` silnice 1. třídy pouze pro auta

V těchto kategoriích pozemních komunikací existují cesty, které umožňují, aby se po nich pohybovali chodci i vozidla – takové cesty můžeme nazvat „obojetné“. Obojetné cesty jsou tyto:

1. místní komunikace 1. třídy
2. místní komunikace 2. třídy
3. místní komunikace 3. třídy
4. silnice 2. třídy
5. silnice 3. třídy

silnice 1. třídy není jednoznačná. Po některých silnicích 1. třídy mohou jezdit jen auta a po jiných mohou jezdit auta i chodit chodci. Toto je vidět obrázku 2. Silnice 35 je pouze pro auta a ulice Zhořelecká je i pro chodce, přitom jsou obě silnicemi 1. třídy. Dále existují cesty, kde mohou chodit jen chodci.

Příklad tvorby konkrétní křižovatky



Obrázek 3 - Příklad tvorby křižovatky. Zdroj: mapy.cz

Na obrázku 3 vidíme výřez z mapy. Bude na něm názorně ukázáno, jak se vytvoří řetězec popisující křižovatku. Budeme řešit křižovatku v černém kroužku.

Nejprve je třeba určit identifikátor (ID) křižovatky a jméno křižovatky.

1. ID – je možno zvolit libovolně, protože se na chodu aplikace nijak nepodílí. Zde zadáme ID rovno *1*.
2. Jméno – to vytvoříme podle názvů ulic vstupujících do křižovatky. Vzhledem k tomu, že se v ní kříží ulice Na Okruhu a Generála Píky, vznikne název ze jmen těchto ulic. Jména křižovatek tedy mohou být *Na Okruhu_Generála Píky* nebo *Generála Píky_Na Okruhu*. Na pořadí názvů jednotlivých ulic nezáleží, proto zvolíme první název.

Poté, co máme definováno ID a jméno křižovatky, je nutné určit vlastnosti sousedních křižovatek vzhledem k právě vytvářené. Tyto vlastnosti jsou název, vzdálenost a typ cesty.

1. Název – při tvorbě názvů sousedních křižovatek postupujeme stejně jako při řešení jména právě tvořené křižovatky.
2. Vzdálenost – je třeba určit vzdálenosti od tvořené křižovatky k sousedním. Ty změříme na nějakém mapovém serveru např. na www.mapy.cz (Mapy.cz).
3. Typ cesty – jde o skutečnou kategorii pozemní komunikace. Ty lze zjistit na adrese mapa.dopravniinfo.cz (Dopravní informace).

   Informace o sousedních křižovatkách jsou následující:

1. soused:

Název: `Na Okruhu_Terronská`, Vzdálenost: `137 m`, Typ cesty: `místní komunikace 3. třídy`

2. soused:

Název: `Žižkovo Náměstí_Generála Píky`, Vzdálenost: `105 m`, Typ cesty: `místní komunikace 2. třídy`

3. soused:

Název: `Generála Píky_Holého`, Vzdálenost: `100 m`, Typ cesty: `místní komunikace 2. třídy`

4. soused:

Název: `Na Okruhu_Údolní`, Vzdálenost: `120 m`, Typ cesty: `místní komunikace 3. třídy`

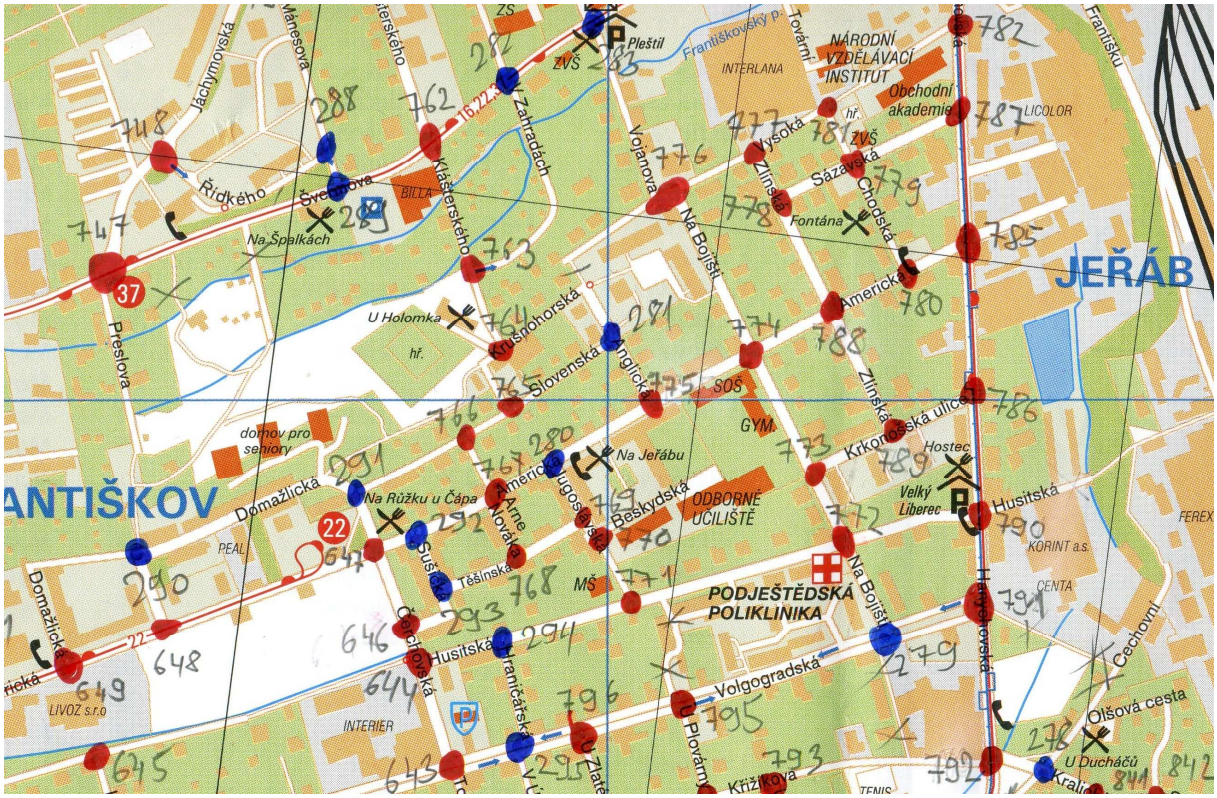
Nakonec je třeba dosadit tyto informace do obecného schématu pro řetězec, který definuje křižovatku (*I*). V našem případě to bude vypadat následujícím způsobem:

```
1+Na Okruhu_Generála Píky*Na Okruhu_Terronská,137,mistniKomunikace3|Žižkovo Náměstí_Generála Píky,105,mistniKomunikace2|Generála Píky_Holého,100,mistniKomunikace2|Na Okruhu_Údolní,120,mistniKomunikace3
```

Takto budou vypadat definice všech křižovatek v datovém souboru pouliční sítě daného města.

4 Tvorba datového souboru

Jak je známo z předchozí kapitoly, je třeba definovat každou křižovatku v Liberci. Abychom eliminovali chybovost v datovém souboru a tím omezili nesprávné výsledky vyhledávání, je dobré při jeho tvorbě používat tištěnou mapu. Mapa nám bude sloužit jako místo, kam si budeme označovat již definované křižovatky. Způsob značení je ukázán zde:



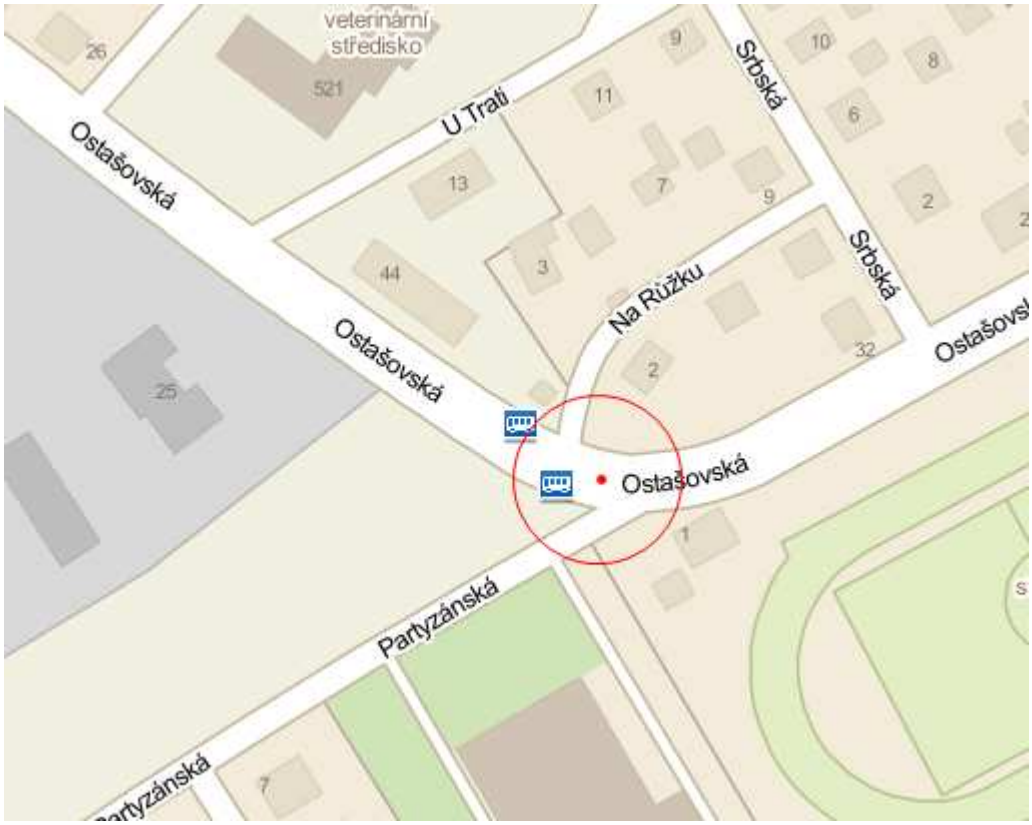
Obrázek 4 - Použitá mapa

Vidíme, že každá definovaná křižovatka je označena tečkou a očíslována. Toto číslo má význam *ID* z kapitoly 3. Číslování slouží k rychlému nalezení dané křižovatky v datovém souboru. Také přispívá k jednoznačnosti názvů křižovatek, která musí být dodržena. Pokud by dodržena nebyla, nefungoval by vyhledávací algoritmus. Na obrázku 4 jsou vidět červené a modré body označující definované křižovatky. Dvě barvy jsou zde z důvodu, abychom nemuseli křižovatky číslovat velkými číslicemi, což by se stalo vzhledem k jejich množství. Prvních tisíc křižovatek je značeno červeně a zbylé křižovatky jsou značeny modře a číslování je opět od jedničky. Čtyřmístná čísla by mohlo být problém zapisovat do mapy na některých místech. V datovém souboru je ale třeba rozlišit jednotlivá ID vzhledem k barevnému značení

na mapě. To je řešeno pomocí písmen, která se připojí k hodnotě ID. Pro červené body je použito písmeno „a“ a pro modré body „b“. V datovém souboru to může vypadat takto:

- Červená křižovatka – 10a
- Modrá křižovatka – 10b

Pro zjednodušení práce při tvorbě datového souboru je možné si usnadnit tuto činnost pomocí slučování křižovatek. Toto sloučení použijeme v případě, kdy jsou od sebe křižovatky vzdálené velmi málo. Jde o vzdálenost okolo patnácti metrů, kdy je lze sloučit. V tomto případě je ale třeba dbát na měření vzdálenosti k sousedním křižovatkám. Je třeba určit jakési střední místo křižovatky a z něj měřit délku. Příklad křižovatek, kde došlo ke sloučení je vidět zde:



Obrázek 5 - Ukázka sloučení křižovatek. Zdroj: mapy.cz

Na obrázku 5 v oblasti označené kružnicí vidíme dvě křižovatky, které je možno sloučit do jedné. Sloučená křižovatka tak získá název *Ostašovská_Partyzánská_Na Růžku*. Červená tečka na obrázku 5 vyznačuje místo, odkud bychom měli měřit vzdálenosti k sousedním křižovatkám, aby byly zachovány skutečné vzdálenosti.

Dalším problémem při tvorbě souboru byla u některých křižovatek nejednoznačnost jejich názvů. V celém Liberci se našlo několik dvojic křižovatek, jejichž názvy podle metodiky tvorby názvů z kapitoly 3, byly stejné. V tomto případě se muselo k názvu jedné přidat něco, co by oba názvy odlišilo. Obvykle se k názvu přidalo jméno určitého objektu v blízkosti křižovatky např. zastávky MHD, obchodního střediska atd. Ukázku nejednoznačnosti názvů vidíme na následujícím obrázku:

Obrázek 6 - Nejednoznačnost názvů křižovatek. Zdroj: mapy.cz

Na obrázku 6 jsou zakroužkované křižovatky *Londýnská_Hokešova*. Tyto dva názvy je třeba od sebe odlišit. Vzhledem k tomu, že u jedné z křižovatek se nachází pneuservis, můžeme ho použít v názvu. Jména obou křižovatek mohou být takováto: *Londýnská_Hokešova*, *Londýnská_Hokešova (u pneuservisu)*.

5 Popis programu

Následuje popis způsobu funkce programu. Podrobný popis funkčnosti celé aplikace lze nalézt v příloze.

5.1 Třída *Graf*

Tato třída obsahuje graf, se kterým aplikace pracuje. Tento graf je zde reprezentován přesně tak, jak je uvedeno v kapitole 2.2, tedy jako seznam sousedů.

Třída obsahuje jedinou veřejnou metodu, která postupně načte celý datový soubor a tyto informace ukládá do datových struktur grafu.

5.2 Třída *PriorityQueue*

Tato třída representuje prioritní frontu, která je použita v Dijkstrův algoritmu. Obsahem třídy je datová struktura ukládající křižovatky a tři veřejné metody.

První metoda s názvem *Enqueue* přidává nový prvek do fronty. Přijímá prioritu a název křižovatky. Prioritou je zde myšleno aktuální nejmenší vzdálenost přijímané křižovatky od startovního bodu.

Další metodou je metoda *Dequeue*, která slouží k vrácení názvu křižovatky s nejmenší prioritou. Metoda také smaže tuto vrácenou křižovatku z fronty.

Třetí metoda je *DecreaseKey*, jejímž úkolem je zmenšit prioritu dané křižovatky. Metoda přijímá aktuální hodnotu priority a křižovatku, u které je nutné tuto prioritu zmenšit.

5.3 Třída *MetodyProComboBox*

Tato třída implementuje metody, které definují chování objektů třídy *ComboBox*. Tyto objekty se v aplikaci používají k zadávání křižovatek a vzhledem k celkovému množství

všech křižovatek je vhodné toto zadávání uživatelsky zpříjemnit. Třída obsahuje celkem tři veřejné metody a všechny přijímají referenci na objekt třídy *ComboBox* a *Graf*.

První metodou je metoda *FillinComboBox*, která zajišťuje naplnění nabídky daného comboboxu patřičnými prvky, což jsou názvy křižovatek. Nabídka comboboxů je plněna pouze těmi křižovatkami, jejichž název obsahuje řetězec, který je napsán v řádku comboboxu.

Druhá metoda se nazývá *Click* a používá se jako obslužná metoda události *Click* u objektů třídy *ComboBox*. Tato událost nastane po tom, co klikneme na kamkoli na objekt třídy *ComboBox*. Metoda způsobí smazání dosavadních položek, vysunutí nabídky a naplnění nabídky názvy křižovatek, které budou vyhovovat textu v řádku comboboxu. Dojde tedy ke spuštění metody *FillinComboBox*.

Poslední metodou je třída *TextChanged*. Ta je obslužnou metodou události TextChanged u objektů třídy *ComboBox*. Tato metoda nastává při změně řetězce v řádce comboboxu. Metoda zajišťuje správné plnění nabídky, pokud uživatel bude psát do řádku. Při psaní je vidět, jak se nabídka postupně zmenšuje a uživateli se tak zužují možnosti výběru. Aplikace se tedy takto snaží uživateli napovědět výčtem křižovatek, které obsahují text v řádce. Metoda funguje podobně jako předchozí metoda *Click* s rozdílem, že úvodní mazání obsahu nabídky při jejím spuštění se omezí pouze na případ, kdy je aktivní řádek comboboxu.

5.4 Třída *FormZadaniKrizovatek*

Třída representuje dialogové okno pro zadávání ignorovaných křižovatek. Tyto křižovatky slouží k tomu, že aplikace přes takto zadané křižovatky nebude vyhledávat nejkratší cestu. Uživatel si tedy může vybrat, které křižovatky zakáže a výsledná cesta je nebude obsahovat.

Konstruktor třídy přijímá reference na hlavní formulář aplikace a objekt třídy *Graf*. Okno obsahuje objekt třídy *ComboBox*, který slouží k zadávání ignorovaných křižovatek. Pro zjednodušení zadávání křižovatek combobox implementuje v obslužných metodách událostí *Click* a *TextChanged* stejnojmenné metody ze třídy *MetodyProComboBox*. Do těchto metod vstupuje reference na daný combobox a na objekt třídy *Graf*.

Dialogové okno dále obsahuje tlačítko *buttonOK*, které zpracuje zadanou křižovatku. Zpracování se provede tak, že křižovatku uložíme do formuláře hlavní aplikace. K tomu nám poslouží reference na tento formulář, kterou jsme přijali v konstruktoru. Nakonec dojde k uzavření tohoto okna.

Posledním prvkem zde je tlačítko *buttonCancel*, kterým smažeme všechny dosavadně zadané křižovatky uložené ve formuláři hlavní aplikace a uzavřeme tento dialog.

5.5 Třída *DijkstruvAlgoritmus*

Třída obsahuje funkčnost pro samotný Dijkstrův algoritmus, který vyhledává nejkratší cestu v síti.

Konstruktor třídy přijímá referenci na objekt třídy *Graf*. Třída deklaruje důležité instanční proměnné. Jde o pole předků s názvem *polePredku*, pole vzdáleností s názvem *poleVzdalenosti* a objekt *fronta* třídy *PriorityQueue*. Třída obsahuje dvě veřejné metody.

První metoda se jmenuje *Inicializace* a slouží k inicializaci datových struktur algoritmu. Metoda přijímá startovní a cílovou křižovatku, které ukládá do instančních proměnných. Metoda vynuluje pole *polePredku*, do pole *poleVzdalenosti* vloží maximální možné hodnoty kromě pozice startovního vrcholu, kam uloží hodnotu nula. Nakonec je potřeba ještě inicializovat objekt *fronta*. K tomu použijeme pole *poleVzdalenosti*, kde máme uloženy aktuální nejkratší vzdálenosti od startovního vrcholu. Ty nám budou sloužit jako priority k vrcholům. Protože vrcholy jsou reprezentovány jako indexy polí, použijeme při plnění prioritní fronty obsah položky pole *poleVzdalenosti* a index této položky. K samotnému přidávání vrcholů do fronty použijeme veřejnou metodu *Enqueue* třídy *PriorityQueue*.

Druhou metodou je metoda *Start*, která představuje Dijkstrův algoritmus. Metoda přijímá čtyři argumenty. Prvním je reference na objekt třídy *ListView*, což je objekt, do kterého se zapisuje výsledek aplikace tedy nejkratší cesta. Druhým je pole *typCesty*, které ukládá aktuální nastavení povolených kategorií pozemních komunikací. Toto pole se v algoritmu používá pro výběr vhodných sousedních vrcholů. Další parametr je reference na objekt *chbPesky* třídy *CheckBox*, který udává, zda je nastaveno režim dopravy „pěšky“ nebo „vozidlem“. Funkčnost metody je stejná, jako je popsána v kapitole 2.3.2.

5.6 Třída *Form1*

Třída *Form1* představuje hlavní formulář aplikace.

Třída obsahuje čtyři instanční proměnné. Jde o deklaraci objektu *graf* třídy *Graf*. Druhou proměnnou je objekt nazvaný *dijkstruvAlg*, který je instancí třídy

DijkstruvAlgoritmus. Dále je zde pole typu *TypCesty* nazvané *typCesty*. Toto pole má velikost šest a slouží k uložení informací, přes které typy kategorií pozemních komunikací je možné vyhledávat nejkratší cestu. Poslední proměnnou je seznam *seznamIgnorovanychKrizovatek*, který slouží k uložení libovolného počtu ignorovaných křižovatek. Tedy těch křižovatek, přes které algoritmus nevyhledává nejkratší cestu.

Konstruktor třídy nejprve spustí metodu *SestrojGraf* objektu *graf* třídy *Graf*. Dále inicializuje objekt *dijkstruvAlg* třídy *DijkstruvAlgoritmus* a do jeho konstruktoru vloží referenci na instanci *graf*.

Metoda *buttonZadatNepovoleneKrizovatky_Click* je obslužnou metodou události *Click* tlačítka *buttonZadatNepovoleneKrizovatky*. Metoda řeší naplnění seznamu *seznamIgnorovanychKrizovatek* křižovatkami, které bude program ignorovat. Naplnění křižovatkami se provede tak, že se spustí vícekrát za sebou objekt třídy *FormZadaniKrizovatek*, což je dialogové okno pro zadávání ignorovaných křižovatek. Kolikrát se má toto okno spustit udává vlastnost *Value* objektu třídy *NumericUpDown*. Uživatel postupně v každém dialogovém okně nastaví danou křižovatku a ta se uloží do instanční proměnné *seznamIgnorovanychKrizovatek*.

Metoda *NastavPoleTypuCest* uloží do pole *typCesty*, zda je daná kategorie pozemních komunikací povolená pro vyhledávání nejkratší cesty. Metoda zjišťuje, jestli je daná komunikace povolená z příslušných objektů třídy *CheckBox*. Pokud povolena je, tak se na příslušnou položku pole uloží odpovídající kategorie pozemních komunikací z výčtového typu *TypCesty*. V případě, že povolena není, uloží se do pole hodnota *nic*.

Metoda *checkBoxPesky_CheckedChanged* je obslužnou metodou události *CheckedChanged* objektu *checkBoxPesky* třídy *CheckBox*. Tento checkbox udává, jestli je vybrán typ dopravy „autem“ nebo „pěšky“. Pokud je instance *checkBoxPesky* zaškrtnutá, pak uživatel nastavil možnost dopravy „pěšky“, v opačném případě je vybrána možnost dopravy „autem“. V těle metody je potřeba zjistit, jestli je objekt *checkBoxPesky* zaškrtnutý nebo není. V případě, že je, se pole *typCesty* naplní všemi hodnotami obojetných typů kategorií pozemních komunikací a hodnotou *silnice1AutaChodci* pro silnici 1. třídy. V případě, že tento checkbox zaškrtnut není, tak se spustí metoda *NastavPoleTypuCest*.

Metoda *checkBox_CheckedChanged* je obslužná metoda události *CheckedChanged* objektů třídy *CheckBox*, které jsou umístěny na obou kontejnerech třídy *GroupBox*. V těle této metody se spustí metoda *NastavPoleTypuCest* ale jen v případě, že objekt *checkBoxPesky* není zaškrtnut.

Metoda *VratPocetZnakuNejdelsihoJmena* vrací počet znaků křižovatky, která má nejdelší název ze všech křižovatek, které jsou výsledkem vyhledávacího algoritmu. V těle metody je inicializována proměnná *delka* typu *int* na hodnotu *-1*. Postupně projdeme všechny křižovatky v komponentě listview. V případě, že nalezneme křižovatku s delším názvem než je hodnota proměnné *delka*, tak počet znaků jejího názvu vložíme do proměnné *delka*. Na konci metody vrátíme proměnnou *delka*, kde bude počet znaků nejdelšího názvu.

Metoda *VratTecky* vrací řetězec teček proměnlivé délky a přijímá dva parametry. První je *delkaNazvu* typu *int*, který obsahuje počet znaků aktuální křižovatky. Druhý je *delkaNejdelsihoNazvu*. Ten je také typu *int* a obsahuje počet znaků nejdelšího názvu křižovatky. Metoda se používá pro export výsledku aplikace do textového souboru. Z důvodu přehlednosti je potřeba, aby křižovatky tvořící nejkratší cestu a vzdálenosti mezi nimi tvořily v souboru dva sloupce. Tato metoda vrací řetězec teček, který má proměnlivou délku v závislosti na rozdílu mezi počty znaků v názvech křižovatek. Konkrétní počet teček je určen rozdílem mezi parametry *delkaNejdelsihoNazvu* a *delkaNazvu*. K tomuto výsledku je ještě přičtena konstanta *2*, která posune o dvě tečky sloupec vzdáleností, aby nebyl nalepený na názvu nejdelší křižovatky.

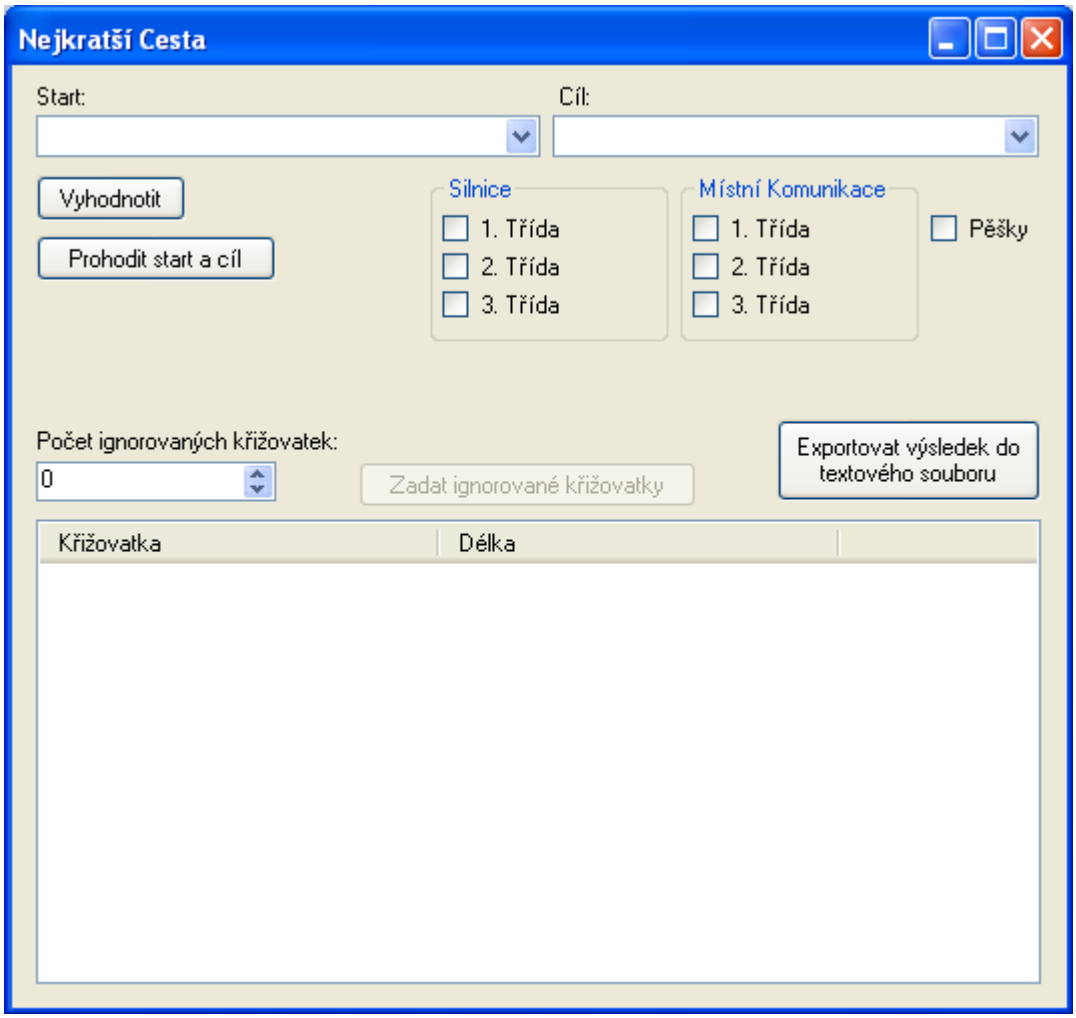
Metoda *buttonExport_Click* je obslužnou metodou tlačítka *buttonExport*, které umožňuje exportovat výslednou nejkratší cestu do textového souboru. Exportovat je možné pouze v případě, že máme vygenerovanou cestu v komponentě listview. Pokud tomu tak je, spustíme metodu *VratPocetZnakuNejdelsihoJmena*, kterou zjistíme počet znaků nejdelšího názvu křižovatky a výsledek uložíme do proměnné *pocetZnakuNejdelsihoJmena*. Objekt *sfd* třídy *SaveFileDialog*, který představuje ukládací dialog, spustíme. Poté vybereme místo uložení a pokud okno zavřeme jinak než kliknutím na tlačítko OK, ukončíme celou exportovací metodu. V opačném případě do souboru zapíšeme povolené kategorie místních komunikací. Dále zrekonstruujeme postupně každý řádek z listview. To provedeme tak, že spojíme obsah buňky prvního sloupce a potřebný počet teček, který vrací metoda *VratTecky* a nakonec připojíme obsah buňky druhého sloupce. Tyto řádky zapíšeme postupně do souboru.

Poslední metodou třídy *Form1* je metoda *buttonVyhodnotit_Click*. Jde o obslužnou metodu události *Click* tlačítka *buttonVyhodnotit*. Metoda zajišťuje výpočet nejkratší cesty vzhledem k zadaným datům. Nejprve je potřeba vymazat dosavadní obsah objektu *listView1* třídy *ListView* pomocí metody *Clear*. V případě, že obě zadané křižovatky existují, spustíme metody *Inicializace*, do které vložíme startovní a cílovou křižovatku a *Start*, kam vložíme objekty *listView1*, pole *typCesty*, checkbox *checkBoxPesky* a seznam

seznamIgnorovanychKrizovatek. Nakonec do vlastnosti *Value* objektu *numericUpDown1* vložíme hodnotu nula.

6 Popis uživatelského rozhraní – ovládání

Na následujícím obrázku můžeme vidět hlavní formulář aplikace. Tento formulář je objektem třídy *Form1*.



Obrázek 7 - Hlavní formulář aplikace

Horní polovina formuláře slouží k zadávání vstupních dat do aplikace a spodní polovina, kde vidíme komponentu *ListView*, slouží jako výstup programu.

6.1 Zadávání křižovatek

Pomocí dvou nahoře umístěných rozklikávatelných řádků tzv. comboboxů je možné nastavit startovní a cílovou křižovatku, mezi kterými bude následně program vyhledávat nejkratší cestu. Při kliknutí na ně se vysune nabídka, která obsahuje všechny křižovatky, které můžeme zadat. Jejich množství závisí na obsahu použitého datového souboru. Můžeme tedy vybranou křižovatku vložit přímo z nabídky anebo její název vepsat do řádku. Pokud budeme psát do řádku, bude se nabídka křižovatek přizpůsobovat psanému textu. To znamená, že se celkové množství křižovatek omezí jen na ty křižovatky, které v názvu obsahují řetězec napsaný v řádku comboboxu. Pokud by uživatel do řádku napsal řetězec, který neobsahuje žádný název křižovatky, tak aplikace tento řetězec smaže. To provede tak, že do řádku vloží prázdný řetězec.

6.2 Nastavení platných kategorií pozemních komunikací

Zaškrtávací políčko (checkbox) s názvem Pěšky udává, jestli uživatel zvolil možnost dopravy „autem“ nebo „pěšky“. Pokud je políčko *Pěšky* zaškrtnuto, je zvolen způsob dopravy „pěšky“. Když zůstane toto políčko nezaškrtnuto, je zvolen typ dopravy „autem“. V případě, že zvolíme způsob dopravy „pěšky“, vnitřní mechanismus aplikace nastaví, aby byly prohledávány všechny křižovatky, k nimž je přístup pro pěší. To znamená, že k těmto křižovatkám vedou některé z obojetných kategorií pozemních komunikací, silnice 1. třídy, která je pro vozidla i pro chodce nebo komunikace, která je pouze pro chodce. Pokud zvolíme způsob dopravy autem, vnitřní mechanismus aplikace nastaví, aby byly prohledávány všechny křižovatky, k nimž je přístup pro auta. Které kategorie pozemních komunikací to konkrétně budou, záleží na uživatelském nastavení aplikace.

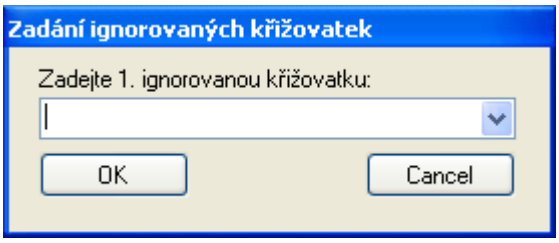
Nastavení provádějící povolení daných tříd kategorií pozemních komunikací se provádí pomocí checkboxů, které jsou ohraničeny popiskou *Silnice* nebo *Místní Komunikace*. Jiné kategorie místních komunikací jako dálnice v Liberci nejsou, proto v aplikaci jsou jen možnosti nastavení tříd silnic a místních komunikací. Program bude vyhodnocovat nejkratší cestu pouze přes povolené třídy komunikací, které jsou zadány uživatelem pomocí zmíněných checkboxů. Pokud zadáme typ dopravy „pěšky“ a zároveň budou některé checkboxy pro silnice a místní komunikace zaškrtnuty, program na dané zaškrtnutí nebere ohled. Nastavení

komunikací bude v tomto případě provedeno tak, aby byly povoleny všechny pozemní komunikace pro chodce. Při odškrtnutí políčka *Pěšky* a tedy výběru typu dopravy „vozidlem“ budou zadány třídy silnic a místních komunikací dle aktuálního nastavení jejich checkboxů.

6.3 Zadání množiny ignorovaných křižovatek

Program umožňuje zadat množinu ignorovaných křižovatek. Tedy takových křižovatek, přes které program nebude vyhledávat nejkratší cestu. Této možnosti využijeme v případě, že chceme najít nejkratší cestu mezi dvěma body, ale chceme se vyhnout daným křižovatkám.

Definování této množiny křižovatek je řešeno dvěma komponentami. První komponentou je číselník s popiskou *"Počet ignorovaných křižovatek:"*, do kterého zadáme počet ignorovaných křižovatek. Druhou je tlačítko s nápisem *"Zadat ignorované křižovatky"*. Tímto tlačítkem zadáváme do programu konkrétní křižovatky. Pokud máme v číselníku zadán počet nula ignorovaných křižovatek, máme tlačítko zešedlé a po kliknutí na něj se nic nestane. Proto je třeba v číselníku nastavit určitou kladnou hodnotu a poté se tlačítko uvede do normálního stavu a umožní tak uživateli s ním pracovat. Po kliknutí na toto tlačítko se otevře dialogové okno pro zadání ignorovaných křižovatek:



Obrázek 8 - Dialogové okno pro zadání ignorovaných křižovatek

V tomto dialogovém okně zadáváme křižovatky stejným způsobem jako v hlavním okně aplikace. Z důvodu, že zde můžeme zadat pouze jednu křižovatku, tak se toto dialogové okno otevře postupně vícekrát. Počet spuštění tohoto okna závisí na hodnotě, kterou jsme zadali do číselníku. Po vybrání dané ignorované křižovatky klikneme na tlačítko *OK*, čímž ji zadáme do programu. Po zadání celé množiny těchto křižovatek se číselník vynuluje a tlačítko pro zadávání ignorovaných křižovatek opět zešedne a změní svůj nápis na *"Ignorované křižovatky zadány"*. Zešednutí tlačítka a tedy odepření možnosti pracovat s tímto tlačítkem je

z důvodu toho, že už nechceme nijak měnit zadanou množinu křižovatek. Pokud by uživatel chtěl takto zadané křižovatky změnit, např. z důvodu chybného zadání, musí je smazat a zadat znovu. Smazání se provede kliknutím na tlačítko Vyhodnotit. Toto tlačítko už umožňuje vyhledat nejkratší cestu. V případě, že v hlavní aplikaci nezadáme žádnou startovní ani cílovou křižovatku, tak tlačítko způsobí pouze smazání množiny ignorovaných křižovatek. Poté se na tlačítku pro zadávání ignorovaných křižovatek objeví nápis *"Zadat ignorované křižovatky"*, čímž se dostaneme do výchozího stavu aplikace. Když by si uživatel uvědomil v průběhu zadávání křižovatek chybu, je zde možnost stisknout tlačítko *Cancel* na dialogovém okně. Kliknutí na toto tlačítko způsobí smazání všech dosavadně zadaných křižovatek a ukončení aktuálního dialogového okna. Tím se aplikace opět dostane do výchozího stavu jako při jejím spuštění.

6.4 Export nejkratší cesty do textového souboru

Program nabízí možnost uložit vyhledanou cestu do textového souboru. K tomuto úkonu slouží tlačítko s názvem „Exportovat výsledek do textového souboru“. Aplikace získává data k zápisu do souboru z komponenty listview. Aby tedy bylo možné cestu exportovat, musí být vyhodnocena a zobrazena v listview. Pokud by listview nezobrazovalo žádnou cestu, pak není co exportovat. Na tuto skutečnost nás aplikace upozorní hláškou "Není vyhodnocena žádná cesta, takže není co exportovat.". Jestliže máme data k uložení do souboru, tak se po kliknutí na exportovací tlačítko otevře klasické windowsovské ukládací dialogové okno. V tomto okně si vybereme, kam chceme soubor uložit a uložíme ho. Protože se jedná o textový soubor, je možné ho uložit pouze s příponou txt.

Způsob zápisu dat do souboru je takový, že se nejprve zapíší povolené kategorie pozemních komunikací, přes které nejkratší cesta vede. Tyto kategorie se zapíší formou platných hodnot z výčtového typu *TypCesty*. Po kategoriích jsou do souboru zapisovány jednotlivé křižovatky včetně vzdáleností mezi nimi. Oba typy informací jsou zapsány do sloupců a jsou odděleny pomocí teček. Použití teček je výhodné v tom, že je jasně vidět, která vzdálenost patří k jaké křižovatce. Na posledním řádku je stejně jako v aplikaci zapsána celková délka vyhledané cesty. Příklad jak vypadá výsledný text v souboru, je možné vidět zde:

```
Cesta prochází přes tyto kategorie pozemních komunikací:
silnice3
silnice1AutaChodci
silnice2
mistniKomunikace3
mistniKomunikace2
mistniKomunikace1

28. Října_Jeronýmova.....................................................0
Doubská (podchod).....................................................71
Doubská_Nákladní_Čechova.....................................................45
Čechova_Šumavská.....................................................135
Šumavská (nájezd na silnici 35).....................................84
Košická_Šumavská.....................................................328
Košická_Nitranská.....................................................191
Dr. Milady Horákové_Košická.....................................113
Dr. Milady Horákové_U Opatrovny.....................................173
Dr. Milady Horákové_Náchodská.....................................62
Dr. Milady Horákové_U Krematoria.....................................183
Dr. Milady Horákové_Poutnická_Na Perštýně_Na Pláni_Lipová_Blažkova..233
Lipová_Moskevská.....................................................68
Oblačná_8. Března_Lipová.....................................95
8. Března_Boženy Němcové.....................................113
Palachova_Felberova_Rumunská_8. Března_Gutenbergova.....................191
Šaldovo Náměstí_Palachova.....................................153
Šaldovo Náměstí_Sokolská_5. Května.....................35
5. Května_Liliová.....................................30
5. Května_Herrmannova.....................................83
5. Května_Voroněžská_V Úvoze_Vzdušná.....................56

Celková vzdálenost.....................................2442 metrů
```

Obrázek 9 - Exportovaná cesta v souboru

V horní části souboru vidíme, že aplikace měla povoleno všech šest kategorií pozemních komunikací. Silnici 1. třídy máme zapsanou jako silnice1AutaChodci, podle čehož se dá usoudit, že aplikace byla nastavena na mód „pěšky“.

6.5 Vyhledání nejkratší cesty

Stiskem tlačítka *Vyhodnotit* způsobíme vyhledání nejkratší cesty mezi křižovatkami zadanými v kolonkách *Start* a *Cíl*. Vlastní vyhledávací algoritmus se spustí pouze v případě, že jsme zadali existující křižovatky pro daný datový soubor. Pokud křižovatky nezadáme nebo zadáme neexistující bod, algoritmus se nespustí, pouze dojde ke smazání množiny ignorovaných křižovatek. Toho lze využít v případě, že uživatel zadal chybně ignorované křižovatky. V takovém případě lze celou množinu smazat kliknutím na tlačítko Vyhodnotit a je možno zadat dané křižovatky znovu. Pokud chceme vyhledávat mezi existujícími křižovatkami a nemáme v aplikaci povolené žádné kategorie místních komunikací, program nás na toto upozorní hláškou "Nejsou nastaveny kategorie pozemních komunikací.". Jestliže jsme po vybrání platných křižovatek nevybrali žádnou z možných kategorií pozemních komunikací, aplikace nás na to upozorní hláškou "Nejsou nastaveny kategorie pozemních

komunikací.". Aplikace obsahuje tlačítko s popiskou "Prohodit start a cíl", která způsobí vzájemnou záměnu zadaných křižovatek v comboboxech. Toto nám umožňuje vyhledat cestu v opačném směru. Tato funkce se hodí zejména pro testování kvality datového souboru, protože obě tyto cesty by měly mít stejné vzdálenosti. Stejná vzdálenost pro tyto cesty je spíše pro režim vyhledávání „pěšky“, protože v režimu dopravy „autem“ mohou nastat problémy s jednosměrnými komunikacemi.

7 Testování aplikace

Zde se budeme věnovat testování naší aplikace a jejímu srovnávání s komerčními produkty jako jsou mapy.cz nebo maps.google.

V programu je možno nastavit jaké kategorie místních komunikací jsou povoleny pro pohyb vozidlem. V našem programu tedy můžeme vidět rozdílný výsledek mezi stejnými křižovatkami, pokud změníme nastavení kategorií pozemních komunikací.

V případě vyhledávání nejkratší cesty mezi křižovatkami *Tržní Náměstí_Rumjancevova_Šamánkova* a *Na Okruhu_Generála Píky* pouze přes místní komunikace 2. třídy anebo pouze přes místní komunikace 3. třídy budou výsledky vypadat takto:

Křižovatka	Délka
Tržní Náměstí_Rumjancevova_Šamánkova	0
Šamánkova_U Náspu	95
Masarykova_5. Května_Šamánkova_Vzdušná	97
Masarykova_Klostermannova	119
Masarykova_U Obchodní Komory	147
Vítězná_Masarykova	93
Masarykova_Dvořákova	145
Dvořákova_Gorkého	113
Dvořákova_Mozartova	90
Zborovská_Dvořákova	96
Údolní_Zborovská	209
Na Okruhu_Údolní	111
Na Okruhu_Generála Píky	120
Celková vzdálenost	1435 metrů

Obrázek 11 - Nejkratší cesta přes místní komunikace 3. třídy

Křižovatka	Délka
Tržní Náměstí_Rumjancevova_Šamánkova	0
Budyšínská_Tržní Náměstí	107
Budyšínská_Durychova	37
Štefánikovo Náměstí_Durychova_Gorkého	236
Štefánikovo Náměstí_Generála Píky_Zboro...	229
Generála Píky_Dvořákova	75
Generála Píky_Holého	55
Na Okruhu_Generála Píky	100
Celková vzdálenost	839 metrů

Obrázek 10 - Nejkratší cesta přes místní komunikace 2. třídy

Na obrázcích 10 a 11 vidíme, že výsledná cesta je rozdílná a celková vzdálenost se velmi liší. Cesta přes místní komunikace 2. třídy je v tomto případě s 839 m kratší oproti cestě přes místní komunikace 3. třídy, která měří 1435 m.

7.1 Srovnání s aplikací Mapy.cz

Aplikace Mapy.cz neumožňuje kategorizovat pozemní komunikace tak jako náš program. Dávají ale možnost zadat několik způsobů dopravy. Jde o tyto: autem, pešky nebo

na kole. Poslední možností je ruční měření vzdálenosti, které při porovnávání obou aplikací nemá smysl. Každý způsob dopravy má další možnosti nastavení, jde o tyto:

1. Autem – lze vyhledat nejkratší anebo nejrychlejší cestu. Pro oba případy zde máme možnost volit, zda chceme, aby výsledek obsahoval placené úseky nebo ne.
2. Pěšky – zde máme možnost preferovat turistické cesty před obyčejnými městskými pozemními komunikacemi.
3. Na kole – zde je možnost upřednostňovat cyklotrasy. Dále můžeme zvolit, jestli se chceme vyhnout se silnicím 1. třídy.

Aplikace dále umožňuje zaměnit startovní a cílový bod a vyhledat cestu v opačném směru. Stejnou možnost obsahuje i náš program. Mapy.cz jsou schopné vyhledat nejkratší cestu přes zadaný bod. Tuto funkci náš program nemá, ale takové chování je možné simulovat způsobem, že vyhledáme cestu mezi startovní a průjezdní křižovatkou a poté mezi průjezdní a cílovou. Výsledkem by byla nejkratší cesta přes danou křižovatku. Ovšem Mapy.cz nejsou schopné vyhledat cestu tak, aby výsledek neobsahoval zadané křižovatky. Tuto schopnost má náš program.

Příklady vyhledávání

1. Vstupní křižovatky: *Okruhu_Generála Píky* a *Hanychovská_Krkonošská*

   Mapy.cz:

   Vzdálenost: 3400 m

   Způsob dopravy: Autem

   Naše aplikace:

   Vzdálenost: 3443 m

   Způsob dopravy: Autem – místní komunikace 2. a 3. třídy

   Nejkratší cesta vyhledaná aplikací Mapy.cz je stejná jako výstup našeho programu. Oba výsledky se dobře shodují.

2. Vstupní křižovatky: *Okruhu_Generála Píky* a *Vojtěšská_Slovenského Národního Povstání*

   Mapy.cz:

   Vzdálenost: 2100 m

Způsob dopravy: Autem

Naše aplikace:

Vzdálenost: 2078 m

Způsob dopravy: Autem – povoleny všechny třídy místních komunikací a silnic

Tyto výsledky mají totožnou cestu a vzdálenosti se také dobře shodují. Výsledky Map.cz pro mód dopravy pěšky a autem se shodují.

3. Vstupní křižovatky: *Hejnická_Generála Svobody_Kateřinská* a *Svobody_Hrubínova*

Mapy.cz:

Vzdálenost: 4900 m

Způsob dopravy: Pešky

Naše aplikace:

Vzdálenost: 4783 m

Způsob dopravy: Pěšky

Výsledné cesty obou aplikací jsou totožné, přes to se výsledky liší o 117 m. Problém je v tom, že aplikace Mapy.cz výslednou vzdálenost zaokrouhluje nahoru. Skutečná délka této cesty podle Map.cz je 4804 m. Tento výsledek vznikl sečtením všech úseků cesty. Z důvodu, že Mapy.cz udávají výslednou délku vždy s přesností stovek metrů, je v tomto případě nutné zaokrouhlit hodnotu na 4900 m. Skutečné výsledky se stejně jako v předchozím případě shodují. Liší se pouze o 21 m.

4. Vstupní křižovatky: *Kunratická_Východní* a *Jizerská_Lyžařská*

Mapy.cz:

Vzdálenost: 3000 m

Způsob dopravy: Autem

Naše aplikace:

Vzdálenost: 2262 m

Způsob dopravy: Autem – povoleny všechny třídy místních komunikací a silnic

Rozdíly ve vyhledaných cestách je možné vidět zde:

Obrázek 12 - Porovnání výsledných cest. Zdroj: mapy.cz

Výsledné cesty se liší v oblasti cest Na Skřivanech, Vlčí Vrch a Dubový Vrch, Hrubínova. Aplikace Mapy.cz vyhledala cestu přes ulici Na Skřivanech, Vlčí Vrch, zatímco náš program zde cestu vedl přes ulice Hrubínova a Dubový Vrch. Tímto naše aplikace dosáhla lepšího výsledku. Cesta vyhledaná pomocí našeho programu je tedy o 738 m kratší.

7.2 Srovnání s aplikací Google maps

Stejně jako Mapy.cz i aplikace Google maps neumožňuje povolit určité kategorie pozemních komunikací. Máme zde na výběr několik způsobů dopravy, jde o tyto možnosti:

1. Autem – umožňuje vynechat dálnice nebo vyhnout se zpoplatněným úsekům. Jiné možnosti jako např. vyhledat nejrychlejší cestu zde chybí.

2. Veřejnou dopravou – zde máme možnost vybrat si čas odjezdu anebo příjezdu, typ veřejné dopravy (tramvaj, metro, autobus, vlak) a charakter cesty. Ten se dá ovlivnit menším množstvím přestupů nebo menším množstvím pěších přesunů.
3. Pěšky – tato možnost nenabízí žádné další upřesňující volby.

Aplikace umožňuje výslednou vzdálenost representovat v mílích nebo kilometrech. Tato možnost v aplikaci Mapy.cz i v našem programu chybí.

Příklady vyhledávání

1. Vstupní křižovatky: *Okruhu_Generála Píky* a *Hanychovská_Krkonošská*

   Google maps:

   Vzdálenost: 3500 m

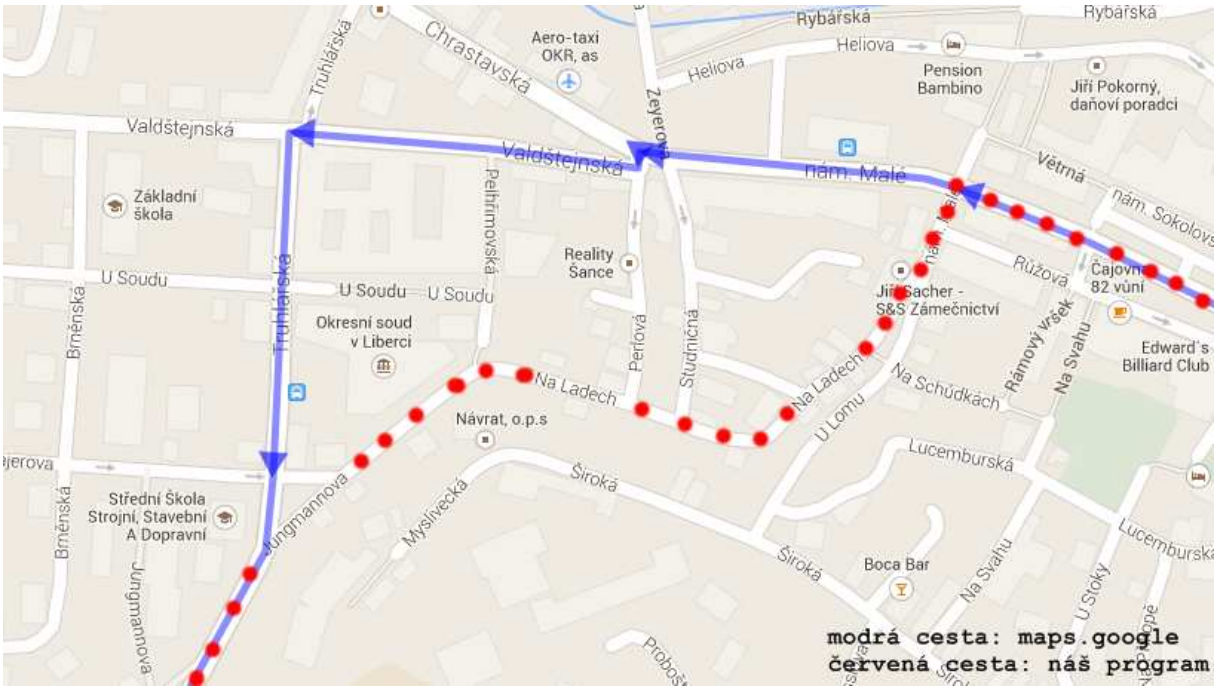
   Způsob dopravy: Autem

   Naše aplikace:

   Vzdálenost: 3443 m

   Způsob dopravy: Autem – místní komunikace 2. a 3. třídy

Rozdíly ve vyhledaných cestách je možné vidět zde:



Obrázek 13 - Porovnání výsledných cest. Zdroj: maps.google.com

Výsledné cesty se liší v oblasti cest Valdštejnská, Truhlářská, Na Ladech a Jungmannova. Aplikace Google maps vyhledala cestu přes ulice Valdštejnská a Truhlářská, zatímco náš program zde cestu vedl přes ulici Na Ladech. Tímto naše aplikace dosáhla lepšího výsledku. Cesta vyhledaná pomocí našeho programu je tedy o 57 m kratší.

2. Vstupní křižovatky: *Hejnická_Generála Svobody_Kateřinská* a *Hodkovická_Cesta JZD*

   Google maps:

   Vzdálenost: 7800 m

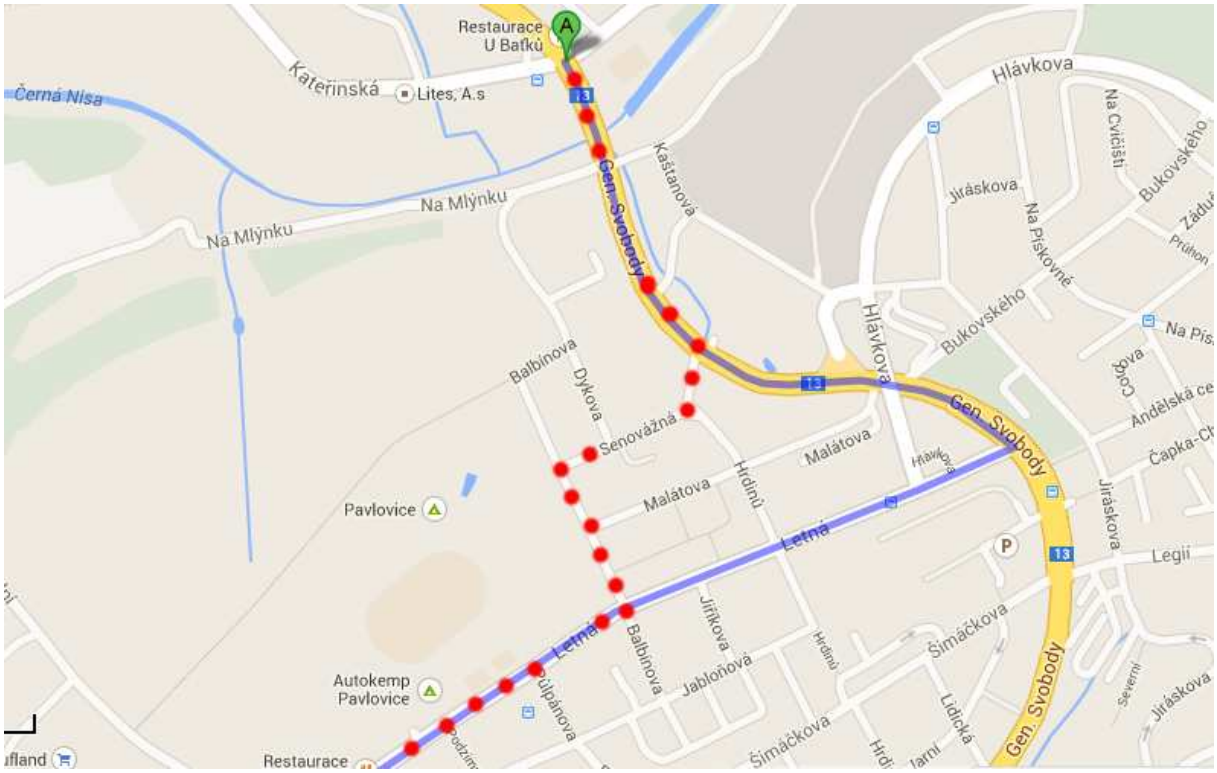
   Způsob dopravy: Autem

   Naše aplikace:

   Vzdálenost: 7357 m

   Způsob dopravy: Autem – povoleny všechny třídy místních komunikací a silnic

Rozdíly ve vyhledaných cestách je možné vidět zde:



Obrázek 14 - Porovnání výsledných cest. Zdroj: maps.google.com

Výsledné cesty se liší v oblasti cest Generála Svobody, Letná, Senovážná a Balbínova. Aplikace Google maps vyhledala cestu přes ulice Generála Svobody a Letná, zatímco náš program zde cestu vedl přes ulice Senovážná a Balbínova. Tímto naše aplikace dosáhla lepšího výsledku, který je o 443 m kratší.

3. Vstupní křižovatky: *Horská_Kateřinská* a *Dubice_Puškinova_Ještědská*

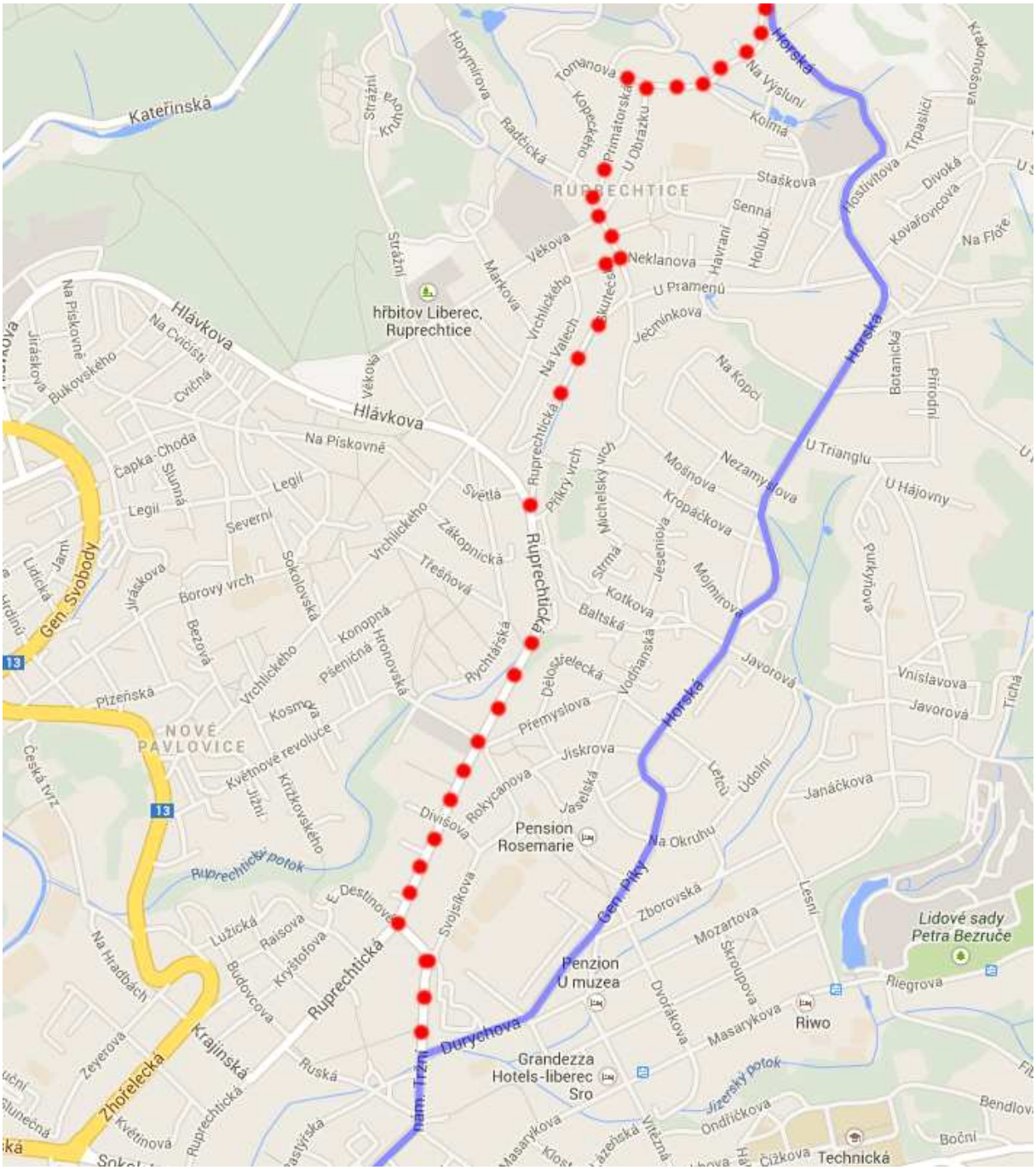
   Google maps:

Vzdálenost:	10,7 km
Způsob dopravy:	Autem

   Naše aplikace:

Vzdálenost:	9877 m
Způsob dopravy:	Autem – povoleny všechny třídy místních komunikací a silnic

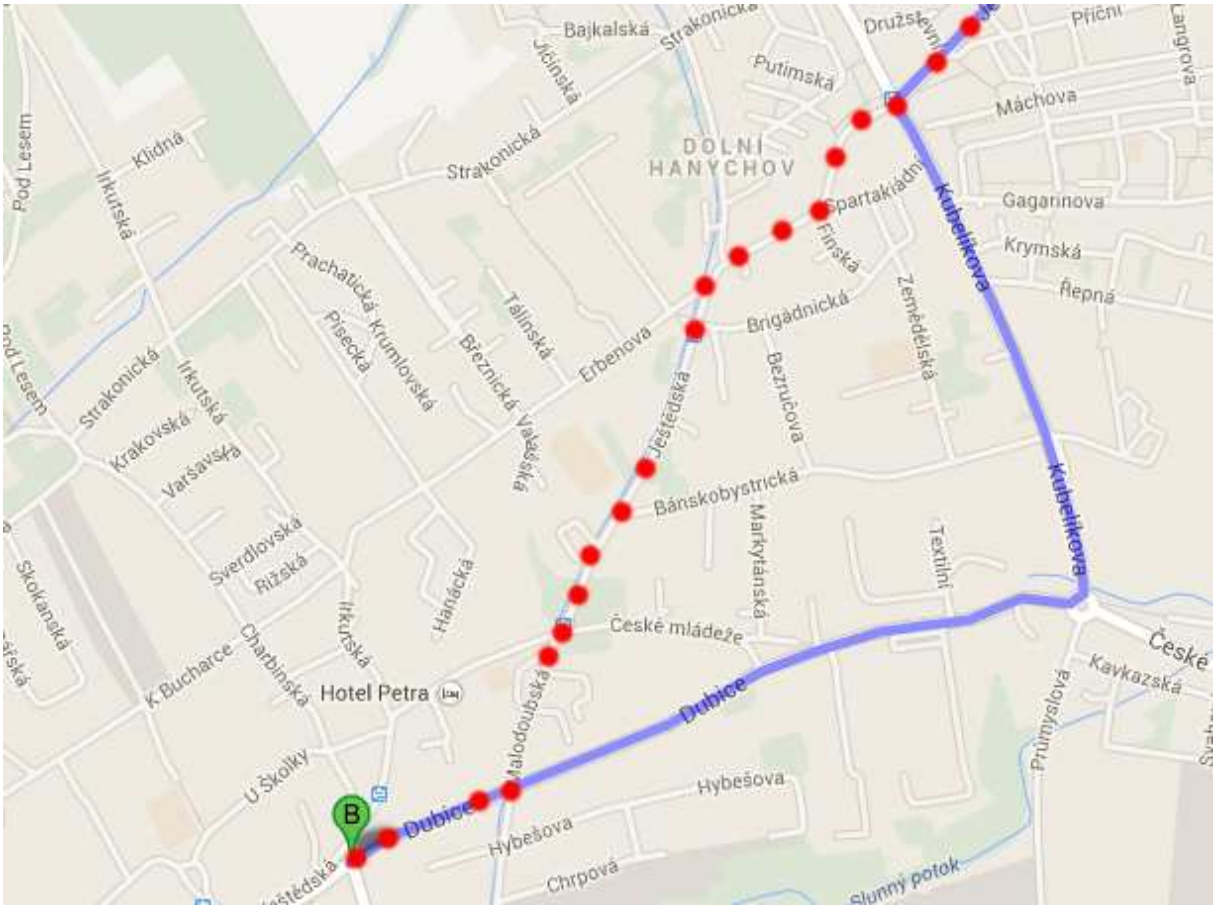
Výsledky hledání mezi těmito křižovatkami se liší celkem na třech místech. První rozdíl je možné vidět na následujícím obrázku:



Obrázek 15 - Porovnání výsledných cest. Zdroj: maps.google.com

Na obrázku 15 je vidět, že aplikace Google maps vede svoji výslednou cestu přes ulici Horská. Celková délka tohoto úseku je 2800 m. Náš program vyhledal kratší úsek tak, že vedl cestu z ulice Horská do čtvrti Ruprechtice, ze kterých se napojil na ulici Ruprechtická. Tento úsek má délku 2670 m a je tedy o 130 m kratší.

Druhý rozdíl je stejný jako první příklad testování aplikace Google maps. Na tomto úseku tedy zkrátíme cestu o 57 m.



Obrázek 16 - Porovnání výsledných cest. Zdroj: maps.google.com

Na obrázku 16 je vidět, že aplikace Google maps vede nejkratší cestu přes ulice Kubelíkova a Dubice. Tento úsek je dlouhý 2000 m. Náš program zde vyhledal cestu přes ulice Ještědská a Malodubská. Délka tohoto kratšího úseku je 1244 m. V této části celkové cesty jsme cestu zkrátili o 756 m.

Celkem je cesta vyhledaná naším programem kratší o 943 m. Pokud tuto hodnotu přičteme celkové délce cesty vyhledané naším programem, vyjde vzdálenost 10820 m. Tato vzdálenost se liší od oficiálního výsledku aplikace Google maps o 120 m. Problém je opět v tom, že tyto mapy celkový výsledek zaokrouhlují na stovky metrů, ale v tomto případě dolů. Skutečná délka cesty vyhledané pomocí aplikace Google maps je 10799 m. To lze zjistit sečtením délek všech úseků cesty. Pokud tedy sečteme délku cesty vyhledanou naším programem a vzdálenost, o kterou jsme si tak ukrátili cestu, vyjde nám hodnota 10820 m. Odsud vidíme, že se délky liší už ne o 120 m, ale jen o 21 m, což je přijatelné.

4. Vstupní křižovatky: *Kunratická_Východní* a *Jizerská_Lyžařská*

   Google maps:

   Vzdálenost: 3500 m

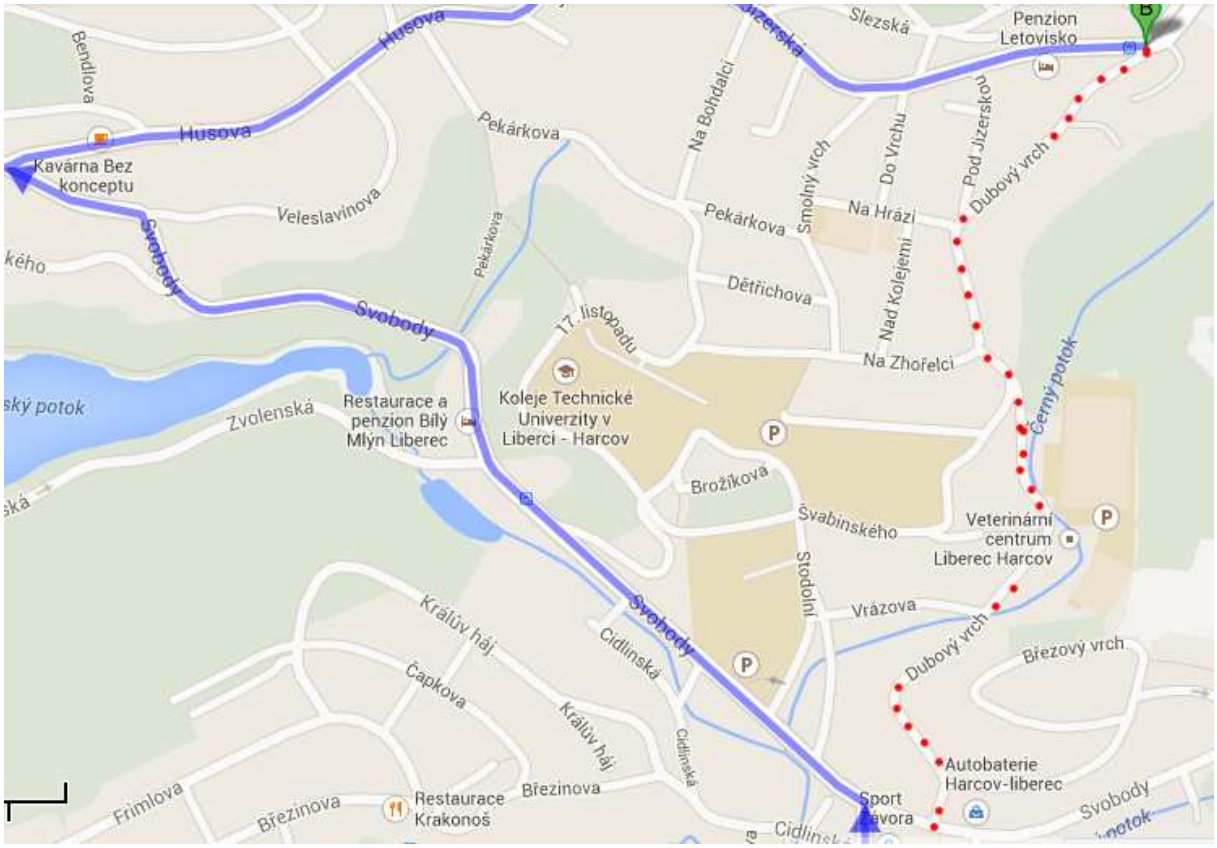
   Způsob dopravy: Autem

   Naše aplikace:

   Vzdálenost: 2262 m

   Způsob dopravy: Autem – povoleny všechny třídy místních komunikací a silnic

Rozdíly ve vyhledaných cestách je možné vidět zde:



Obrázek 17 - Porovnání výsledných cest. Zdroj: maps.google.com

Výsledné cesty se liší v oblasti cest Svobody, Husova, Jizerská a Dubový Vrch. Aplikace Google maps vyhledala cestu přes ulice Svobody, Husova, Jizerská, zatímco náš program zde nejkratší cestu vedl přes ulice Svobody a Dubový Vrch. Tímto naše aplikace dosáhla lepšího výsledku, který je o 1238 m kratší.

8 Závěr

Cílem této bakalářské práce bylo vytvořit software pro vyhledávání nejkratší cesty v pouliční síti města Liberec.

Nejprve bylo nutné se seznámit s tím, jakým způsobem se vyhledává nejkratší cesta a jak je možné representovat městskou síť tak, aby bylo možné v ní tuto cestu vyhledávat. Dále bylo třeba určit, čím bude městská síť representována a jakým způsobem bude vytvořena. Dále bylo třeba obeznámit se, jakým způsobem je možné tuto síť reprezentovat v počítači.

Při psaní programu a vytváření městské sítě bylo potřeba dát pozor na to, aby byly splněny požadavky, které jsou na software kladeny. Jedná se o to, že každá cesta v síti musí mít přiřazenou určitou kategorii pozemních komunikací, která musí odpovídat skutečnosti. Aplikace totiž umožňuje vyhledávat nejkratší cestu jen pro určité uživatelem nastavené kategorie pozemních komunikací. Dále musí mít každá cesta svou délku, aby aplikace mohla získávat údaje o vzdálenosti, které bude používat při vyhledávání nejkratší cesty. Také bylo nutné určit způsob, jak se budou jednotlivé křižovatky pojmenovávat. Program dále umožňuje exportovat výslednou cestu do textového souboru.

Bylo nutné naprogramovat Dijkstrův algoritmus, který je zde použit pro vyhledávání nejkratší cesty. Dále algoritmus, který načítá údaje o městské síti z datového souboru a vytváří z nich datovou strukturu v paměti počítače. Také bylo potřeba naprogramovat způsob zadávání vstupních křižovatek do aplikace. Dalším algoritmem byl např. algoritmus zapisující výslednou nejkratší cestu do textového souboru.

Výsledky našeho programu jsou lepší než výsledky poskytující aplikace Mapy.cz i Google maps.

Další vývoj tohoto softwaru by se mohl zaměřovat na implementaci grafického uživatelského rozhraní. Tedy na to, aby se výsledek vykreslil pomocí čar na načtenou mapu.

9 Použitá literatura

1. ČERNÝ, Jakub. *Základní grafové algoritmy* [online]. c2010 [cit. 2013-08-28]. Dostupné z: kam.mff.cuni.cz/~kuba/ka
2. Dopravní informace. ŘEDITELSTVÍ SILNIC A DÁLNIC ČR. *Dopravní info* [online]. c2009-2010 [cit. 2013-09-05]. Dostupné z: http://mapa.dopravniinfo.cz
3. Mapy.cz. SEZNAM.CZ, a.s. *Seznam* [online]. [cit. 2013-09-05]. Dostupné z: http://www.mapy.cz/
4. Mapy Google. GOOGLE INC. *Google* [online]. c2013 [cit. 2013-09-05]. Dostupné z: https://maps.google.com/
5. MAREŠ, Martin. *Krajinou grafových algoritmů: průvodce pro středně pokročilé*. Vyd. 1. Praha: ITI, 2007, 71 s. ISBN 978-80-239-9049-2.
6. NEŠETŘIL, Jaroslav a Jiří MATOUŠEK. *Kapitoly z diskrétní matematiky*. 1. vyd. Praha: Matfyzpress, c1996, 344 s. ISBN 80-858-6317-0.

10 Příloha – Podrobný popis funkčnosti tříd aplikace

Následuje popis způsobu funkce programu. Bude zde popsána činnost všech tříd aplikace.

10.1 Popis použitých datových struktur

List<> – jedná se o seznam. Ten funguje jako dynamické pole. Můžeme tedy za běhu programu přidávat nebo odebírat jednotlivé položky. Položky zde mají význam vrcholů grafu, v našem případě jde o křižovatky městské sítě. Pro přidávání těchto prvků slouží metoda *Add* a pro jejich mazání jsou zde metody *Remove*, *RemoveAt*, *RemoveAll* a *RemoveRange*. K obsahu položek se přistupuje pomocí indexů, je to tedy stejné jako u pole. Jedná se o generickou třídu, proto je třeba při tvorbě objektu této třídy určit datový typ do lomených závorek. Pak je možné do seznamu vkládat pouze objekty daného typu.

Dictionary<key, value> - jedná se o slovník. Jde opět o dynamickou strukturu, která je generická. Při tvorbě objektu tohoto je potřeba určit dva datové typy. Položky, které slovník obsahuje, zde mají význam křižovatek městské sítě. Je opět možné položky přidávat metodou *Add* a mazat metodou *Remove* stejně jako u seznamu, ale je zde rozdíl. Přidávaná položka tvoří dvojici *key* a *value* a nefunguje zde indexace jako u seznamu. K položkám *value* se přistupuje pomocí hodnoty *key* a to tak, že se za název slovníku napíší hranaté závorky a do nich se vloží hodnota *key*. Funguje to jako indexace v klasickém poli s tím rozdílem, že slovníky se dají indexovat jakýmkoli datovým typem.

SortedList<key, value> - jedná se o seznam, který je schopen sám třídit položky *value* podle hodnoty *key*. Položky této datové struktury v této práci mají opět význam křižovatek. Nové prvky přidáváme metodou *Add*, do které vstupují jako argumenty hodnoty *key* a *value*. Hodnota *key* zde funguje jako priorita, podle které struktura třídí hodnoty *value*. Mazání prvků se provádí metodami *Remove* a *RemoveAt*. K položkám *value* je možné přistupovat pomocí hodnoty *key* anebo pomocí klasické indexace jako v poli nebo struktuře *List*.

10.2 Třída Graf

Úkolem této třídy je vytvořit a uchovat v datových strukturách graf městské sítě. Třída má tyto instanční proměnné:

seznamKrizovatek – je seznam (*List*) typu *int*. Tento seznam má tentýž význam jako pole *Seznam křižovatek* v kapitole 2.2. Určuje tedy meze sousedních vrcholů (křižovatek) v seznamu sousedů pro daný vrchol grafu čili křižovatku.

seznamSousedu – je seznam (*List*) typu *string*. Tento seznam má stejný význam jako pole *Seznam sousedů* v kapitole 2.2. Obsahuje tedy sousední vrcholy (křižovatky) pro všechny vrcholy v grafu městské sítě.

seznamDelek – je seznam (*List*) typu *double*. Tento seznam je definován souběžně se seznamem *seznamSousedu*. Ukládáme sem vzdálenosti sousedních vrcholů od daného vrcholu pro celý graf městské sítě.

seznamTypu – je seznam (*List*) typu *TypCesty*. *TypCesty* je výčtový typ definující kategorii pozemních komunikací. Zde ukládáme konkrétní typ cesty mezi vrcholy. Opět je tento seznam definován souběžně jako seznam *seznamSousedu*.

To, že jsou seznamy definovány souběžně, znamená, že na stejném indexu v seznamech *seznamSousedu*, *seznamDelek* a *seznamTypu* najdeme dané informace o tomtéž sousedovi.

krizovatkyKeyIsInt<key, value> - tento slovník ukládá pořadové číslo *key* křižovatky a samotný název křižovatky *value*. Pokud máme pořadové číslo křižovatky a chceme ho převézt na název křižovatky, vyhledáme ve slovníku hodnotu *value* na daném pořadovém čísle, které má význam hodnoty *key*. Načteme tedy hodnotu *value* na indexu *key*.

krizovatkyKeyIsString<key, value> - tento slovník ukládá název křižovatky (*key*) a pořadové číslo (*value*). Pokud máme název křižovatky a chceme ho převézt na pořadové číslo křižovatky, vyhledáme ve slovníku hodnotu *value* na daném nazvu křižovatky, který má význam hodnoty *key*. Načteme tedy hodnotu *value* na indexu *key*.

Protože je někdy potřeba pracovat s vrcholy, které jsou chápány jako názvy křižovatek a jindy jako pořadová čísla, jsou zde zavedeny tyto dva slovníky, které transformují interpretaci vrcholu.

Všechny instanční proměnné jsou privátní. Z toho důvodu třída obsahuje veřejnou vlastnost pro každou proměnnou. Vlastnosti jsou jen pro čtení, což znamená, že nelze nijak měnit obsah proměnné, se kterým pracuje. Tyto vlastnosti pouze vrací referenci na danou proměnnou.

Třída Graf má jen jednu veřejnou metodu. Tou je metoda *SestrojGraf*, která má za úkol vytvořit v paměti počítače graf městské sítě definované v datovém souboru. Nejdříve

vytvoříme datový proud mezi naším datovým souborem a aplikací, abychom mohli načítat informace. Dále vytvoříme objekt *seznamSousednichKrizovatek* typu *List<string[]>*, kam budeme ukládat údaje o sousedních křižovatkách a proměnnou *index*, která bude ukládat pořadová čísla křižovatek. Pak následuje zpracování jednotlivých řádků (definic křižovatek) datového souboru. K tomu je použit *while* cyklus. Načtený řádek je třeba rozdělit podle rozdělovacích znaků (+ * | ,). Nejprve je třeba řádek dělit znaky „+“ a „*“, abychom získali název křižovatky, který uložíme s daným pořadovým číslem do slovníků *krizovatkyKeyIsInt* a *krizovatkyKeyIsString*. Poté budeme dělit znakem „|“, čímž získáme řetězec informací o každé sousední křižovatce. Tyto řetězce musí být ještě rozděleny znakem „,“, čímž vznikne pole řetězců (*string[]*), které vložíme do seznamu *seznamSousednichKrizovatek*. Tím jsme získali informace o všech sousedech dané křižovatky ve formě řetězce. Teď je možné plnit datové struktury tvořící graf. Před tím je ale nutné některé údaje přetypovat z řetězce na patřičné datové typy. To se týká vzdálenosti (ze *string* na *double*) a typu cesty (ze *string* na *TypCesty*). Těmito daty už je možno naplnit graf. Po načtení všech řádků je třeba zrušit datový proud mezi aplikací a souborem, to uděláme metodou *Close*.

10.3 Třída PriorityQueue

Tato třída je hlavním stavebním kamenem při tvorbě Dijkstrova algoritmu, který se používá pro vyhledávání nejkratší cesty v grafu.

Třída tvoří prioritní frontu. Tato fronta je odlišná od klasické fronty (FIFO). Každý prvek v prioritní frontě si drží údaj o prioritě. Do fronty tedy vkládáme jednak prioritu a dále samotný prvek. Odlišnost mezi prioritní frontou a FIFO je v tom, že při odebírání z fronty se vybírají prvky na základě priority. Existuje minimální a maximální prioritní fronta. Z minimální fronty se odebírají prvky s minimální prioritou. Z maximální fronty se odebírají prvky s maximální prioritou. Prvky ve frontě mohou být seřazeny podle toho, jak byly vloženy. V tom případě bychom při odebírání musely všechny prvky prohledat a odebrat ten s danou prioritou. Také je možné při vkládání nového prvku datovou strukturu reprezentující prioritní frontu ihned setřídit. Potom bychom při odebírání prvku vybrali vždy některý z krajních prvků podle toho, jestli máme minimální nebo maximální prioritní frontu.

Základem prioritní fronty je objekt *sortList* třídy *SortedList*. Ten je vhodný z důvodu automatického třídění prvků, které jsou v něm obsaženy. Prvky jsou řazeny vzestupně, takže

prvek s nejmenší prioritou se vždy nachází na indexu *0*. Toho je v programu využito, protože potřebujeme minimální prioritní frontu.

Třída obsahuje vlastnost *Delka*, která vrací aktuální počet prvků ve frontě. Počet prvků se zjišťuje voláním vlastnosti *Count* objektu *sortList*.

Dále třída obsahuje metodu *Enqueue*, která zajišťuje přidávání nových prvků do fronty. Do metody vstupují proměnné *key* a *value*, které jsou přidány do objektu *sortList* metodou *Add*.

Další metodou této třídy je metoda *Dequeue*. Ta má za úkol odebírat prvek s nejmenší prioritou. Odebírání je tvořeno dvěma kroky. Nejprve se uloží hodnota value s nejnižší prioritou, pak se celá tato položka smaže a nakonec je vrácena předem uložená hodnota s nejnižší prioritou.

Poslední metodou třídy *PriorityQueue* je metoda *DecreaceKey*. Jejím úkolem je zmenšit hodnotu *key* pro určitou hodnotu *value*. Do metody vstupuje proměnná *newKey*, která představuje nový klíč a proměnná *value*, která určuje hodnotu *value* ve frontě, jejíž klíč *key* se má zmenšit. Nejprve je potřeba vyhledat, kde v objektu *sortList* se nachází prvek s danou hodnotou *value*. To se provede pomocí *for* cyklu, kterým projdeme všechny prvky *sortListu* a porovnáváme hodnoty *value* z fronty s hodnotou *value*, která vstoupila jako argument. Pokud dojde k rovnosti, našli jsme prvek, kde je potřeba zmenšit klíč. Ve skutečnosti zde nejde o doslovné zmenšení daného klíče. To proto, že se nalezený prvek vymaže metodou *RemoveAt*. Tato metoda dokáže smazat prvek na určitém indexu, proto do ní vložíme jako argument index nalezeného prvku. Po smazání prvku vložíme do fronty nový prvek složený z vstupních argumentů metody *DecreaseKey*. K vložení použijeme námi naprogramovanou metodu *Enqueue*.

10.4 Třída MetodyProComboBox

Třída implementuje funkčnost pro komponentu *ComboBox*. Metody v ní definované slouží k tomu, aby ovládání *ComboBoxů* pro zadávání křižovatek do programu bylo uživatelsky příjemné.

Comboboxy se chovají tak, že po jejich rozkliknutí vypíší výčet všech křižovatek. Pokud budeme psát do příkazové řádky *ComboBoxu*, výčet vypsaných křižovatek se bude omezovat. Toto omezení bude takové, že se vypíší jen ty křižovatky, které v názvu obsahují řetězec napsaný v příkazové řádce *ComboBoxu*.

Třída je statická, což znamená, že všechny její položky musí být také statické. Položkami jsou myšleny metody a instanční proměnné. Statická třída nelze použít k tvorbě objektů. K položkám třídy se přistupuje ne přes jméno její instance, ale pouze přes jméno dané třídy. Pokud budeme mít statickou třídu jménem *StatickaTrida* s metodou *Metoda*, bude volání metody *Metoda* vypadat takto:

```
StatickaTrida.Metoda();
```

Statické třídy mají tedy spíše charakter knihovny. Známé třídy v jazyce C#, které jsou statické, jsou např. *Console* nebo *Math*. Třída *Console* slouží k obsluze konzolových aplikací (výpis na obrazovku, načtení znaky z klávesnice…). Třída *Math* obsahuje matematické funkce (sin, cos…).

Objekt třídy *ComboBox* má užitečnou vlastnost *SelectedIndex*. Tato vlastnost vrací index aktuálně vybrané položky. První položka má index *0*, indexace je tedy stejná jako u běžného pole. Pokud nemám vybranou žádnou položku, vrací tato vlastnost hodnotu *-1*. Tato vlastnost je vhodná pro zjištění toho, jestli do *ComboBoxu* píšeme nějaký text anebo vybíráme danou položku. Když budeme vpisovat text, tak vlastnost bude vracet hodnotu *-1*.

Důležitou metodou této třídy je metoda *FillinComboBox*. Jejím úkolem je přidávat do komponenty *ComboBox* jednotlivé křižovatky. Přidávání funguje pro všechny křižovatky, ale pouze po splnění podmínky. Musí být splněno, že text v příkazovém řádku *ComboBoxu* je obsažen v názvu aktuální křižovatky a zároveň nesmíme mít vybranou žádnou položku *ComboBoxu*. To zjistíme podle aktuální hodnoty vlastnosti *SelectedIndex*. V případě, že by žádná křižovatka nevyhovovala a zároveň by byla vysunuta nabídka *ComboBoxu*, vložíme do vlastnosti Text *ComboBoxu* prázdný řetězec. Po přidání všech vyhovujících křižovatek nastavíme kursor v příkazovém řádku *ComboBoxu* na konec řetězce pomocí metody *Select*. Metoda *Select* neslouží přímo k tomuto účelu. Jejím úkolem je vybrat část řetězce v příkazové řádce *ComboBoxu*. Metoda přijímá dva parametry. První parametr nastavuje pozici kursoru v řetězci a druhý určuje, kolik znaků za kursorem bude vybráno. Pokud tyto parametry budou mít stejnou hodnotu, nebude vybrán žádný znak a kursor bude na dané pozici. Hodnota parametrů je rovna délce řetězce (počtu znaků) v příkazové řádce. Délku zjistíme voláním vlastnosti *Length*, což je vlastnost třídy *String*.

Další metoda je *Click*. Ta zajišťuje obsluhu komponenty *ComboBox* po tom, co nastala událost *Click*. Tato událost nastane po tom, co klikneme na kamkoli na objekt třídy *ComboBox*. Metoda smaže všechny položky v *ComboBoxu* tak, že spustí metodu *Clear*. Následně objekt *ComboBox* vysune nabídku nastavením vlastnosti *DroppedDown* na hodnotu *true*. Poté je spuštěna metoda *FillinComboBox*.

Poslední metodou je metoda *TextChanged*. Ta zajišťuje obsluhu komponenty *ComboBox* po tom, co nastala událost *TextChanged*. Tato událost nastane po tom, co změníme text v příkazové řádce komponenty *ComboBox*. Metoda opět maže všechny položky v *ComboBoxu* jako metoda *Click*, ale pouze v případě, že nemáme vybranou žádnou položku *ComboBoxu*. To zase zjistíme podle hodnoty vlastnosti *SelectedIndex*. Poté se spustí metoda *FillinComboBox*.

10.5 Třída FormZadaniKrizovatek

Tato třída představuje formulář pro zadání křižovatky, kterou bude program ignorovat. Aplikace totiž umožňuje uživateli, aby si zvolil daný počet křižovatek, přes které algoritmus nebude vyhledávat. Pomocí tohoto dialogového okna je možné zadat jen jednu křižovatku. Aby bylo možné do programu zadat libovolný počet zakázaných křižovatek, je třeba tento dialog spustit vícekrát za sebou.

Třída obsahuje reference na objekt *graf* a objekt *form*. *graf* je objekt třídy *Graf* a *form* je objekt třídy *Form1*. Objekt *form* představuje referenci na hlavní formulář aplikace. Objekt graf representuje konkrétní model městské sítě.

Konstruktor třídy přijímá reference na objekty *graf* a *form* a celočíselnou hodnotu *i* typu *int*. Proměnná *i* může nabývat hodnot *1 – n*, kde *n* je počet ignorovaných křižovatek. Hodnota *i* je přijímána proto, abychom na formuláři mohli zobrazovat následující řetězec: *"Zadejte {0}. ignorovanou křižovatku:"*. Tento řetězec se vkládá do vlastnosti *Text* objektu *label1* třídy *Label*. Za znak *{0}* se vloží hodnota proměnné *i*. Můžeme tedy vidět, kolikátou ignorovanou křižovatku v pořadí zadáváme. Dále konstruktor uloží přijímané reference na objekt *graf* a *form* do stejnojmenných instančních proměnných.

Dialogové okno obsahuje několik komponent. Jde o jeden objekt třídy *ComboBox* a dva objekty třídy *Button*, což jsou tlačítka.

Pro objekt třídy *ComboBox* je zde metoda *comboBoxIgnorovanaKrizovatka_Click*. Jde o obslužnou metodu události *Click* objektu *comboBoxIgnorovanaKrizovatka*. Metoda vytvoří proměnnou typu *ComboBox* a uloží do ní argument *sender*, který vstupuje do metody. Objekt *sender* je typu *object* a proto je třeba ho přetypovat na typ *ComboBox*. Následně je spuštěna metoda *ComboBoxClick* statické třídy *MetodyProComboBox*. Této metodě předáme jako parametr objekt třídy *ComboBox* a objekt *graf*.

Poslední metodou pro objekt třídy *ComboBox* je metoda *comboBoxIgnorovanaKrizovatka_TextChanged*. Jde o obslužnou metodu události *TextChanged* objektu *comboBoxIgnorovanaKrizovatka*. Metoda vytvoří proměnnou typu *ComboBox* a uloží do ní argument *sender*, který vstupuje do metody. Objekt *sender* je typu *object* a proto je třeba ho přetypovat na typ *ComboBox*. Následně je spuštěna metoda *ComboBoxTextChanged* statické třídy *MetodyProComboBox*. Této metodě předáme jako parametr objekt třídy *ComboBox* a objekt *graf*.

Dále zde máme metodu *buttonOK_Click*. Jde o obslužnou metodu události *Click* tlačítka *buttonOK*. Metoda vloží do vlastnosti *SeznamIgnorovanychKrizovatek* vlastnost *Text* objektu *comboBoxIgnorovanaKrizovatka*. Vlastnost *SeznamIgnorovanychKrizovatek* je veřejný člen třídy *Form1* a způsobuje ukládání zde vybraných křižovatek do určité datové struktury ve třídě *Form1*. Přiřazení vybrané křižovatky do vlastnosti *SeznamIgnorovanychKrizovatek* nastane jen tehdy, pokud je ve vlastnosti *Text* objektu třídy *ComboBox* název existující křižovatky. Existenci názvu křižovatky zjistíme tak, že zkontrolujeme, jestli je uživatelem zadaný řetězec obsažen v některém z objektů třídy *Dictionary*. Zde je vybrán slovník *KrizovatkyKeyIsString*. Pomocí metody *ContainsKey*, která vrací boolovskou hodnotu, zjistíme, zda je daný řetězec obsažen ve slovníku a tedy jestli je název platný. Pokud metoda *ContainsKey* vrátí hodnotu logická jednička, zadaný název je platný a je možné ho přidat do objektu *seznamIgnorovanychKrizovatek*. Do podmínky spadá i spuštění metody *Close*, která způsobí zavření dialogového okna.

Poslední metoda je *buttonCancel_Click*. Jde o obslužnou metodu události *Click* tlačítka *buttonCancel*. Ta způsobí smazání datové struktury ukládající zde vybrané ignorované křižovatky. Smazání provedeme tak, že do vlastnosti *SeznamIgnorovanychKrizovatek* vložíme hodnotu *null*. Tento samotný úkon nezpůsobí smazání dané datové struktury. To způsobí automatická správa paměti jazyka C#. Tato správa maže všechny oblasti paměti počítače, které náleží programu a na které neukazuje žádná reference z aplikace. Pokud tedy při zadávání zakázané křižovatky klikneme na tlačítko *buttonCancel*, tak zrušíme všechny dosavadně zadané křižovatky. Nakonec pomocí metody *Close* zavřeme toto dialogové okno.

10.6 Třída DijkstruvAlgoritmus

Tato třída implementuje samotný Dijkstrův algoritmus. Ten se používá pro vyhledávání nejkratší cesty v grafu. Graf v našem případě popisuje městskou síť.

10.6.1 Instanční proměnné třídy

Třída obsahuje několik instančních proměnných. První proměnnou je reference na objekt *graf* třídy *Graf*. Do této proměnné uložíme náš graf reprezentující model města.

Další důležitou datovou položkou je objekt *fronta*. Tento objekt je popsán třídou *PriorityQueue*. Objekt fronta představuje prioritní frontu. Ta je použita v Dijkstrově algoritmu.

Dále deklarujeme pole, do kterého budeme ukládat jeden z výsledků Dijkstrova algoritmu. Pole je typu *string* a je nazváno *polePredku*. Toto pole slouží k ukládání předků daných vrcholů. Protože v aplikaci máme slovníky umožňující převod interpretace vrcholů grafu (křižovatek) mezi jejich názvem a pořadovým číslem, je vhodné je teď využít. Indexy pole *polePredku* budeme chápat jako vrcholy representované jejich pořadovým číslem. Do jednotlivých buněk tohoto pole budeme ukládat předky daného vrcholu, které budou representovány názvem křižovatky.

Máme zde další pole, které je typu *double* a jmenuje se *poleVzdalenosti*. Do něho uložíme další část výsledků z Dijkstrova algoritmu. Toto pole obsahuje na každé položce nejkratší možnou vzdálenost mezi startovní křižovatkou a danou křižovatkou. Na indexu startovní křižovatky bude uložena vždy nulová hodnota, protože startovní vrchol není sám od sebe nijak vzdálen. Opět v případě tohoto pole používáme stejný způsob interpretace vrcholů jako v předchozím poli tedy v poli *polePredků*.

Dalšími proměnnými jsou proměnné *start* a *cil*. Obě tyto proměnné jsou typu *string*. Ukládáme do nich název startovní a cílové křižovatky, mezi kterými chceme najít nejkratší cestu.

Poslední instanční proměnnou této třídy je objekt *cesta*. Tento objekt je typu *List<string>* a slouží k ukládání křižovatek, které tvoří nejkratší cestu včetně obou krajních křižovatek tedy startovního a cílového vrcholu.

10.6.2 Metody třídy

Konstruktor třídy s názvem *DijkstruvAlgoritmus* přijímá referenci na objekt třídy *Graf*. V těle se provede přiřazení reference na objekt *graf* do instanční proměnné. Dále se inicializují obě pole *polePredku* a *poleVzdalenosti* na velikost odpovídající celkovému počtu křižovatek městské sítě.

Metoda *Inicializace* způsobí inicializaci datových struktur, které používá Dijkstrův algoritmus. Metoda přijímá startovní a cílovou křižovatku, které uloží do instančních proměnných *start* a *cil*. Na každou položku pole *polePredku* vložíme hodnotu *null* a na každou položku pole *poleVzdáleností* maximální možnou hodnotu typu *double*. Do pole *poleVzdalenosti* na index startovního vrcholu je třeba uložit hodnotu nula. Pak už zbývá naplnit prioritní frontu. Tu naplníme pomocí veřejné metody *Enqueue* objektu *fronta*, kterou umístíme do *for* cyklu. *For* cyklus jde přes celkový počet vrcholů grafu. Postupně po fronty vložíme všechny dvojice priorita (*key*) a jméno křižovatky (*value*). Prioritu získáme v poli *poleVzdalenosti* a názvy ve slovníku *KrizovatkyKeyIsInt*.

Metoda *RekonstrukceCesty* má za úkol naplnit seznam cesta křižovatkami, které tvoří nejkratší cestu. Do metody vstupuje proměnná s názvem *krizovatka*. Při použití metody do ní vstupuje cílová křižovatka. Cestu rekonstruujeme tak, že se pomocí rekurze pohybujeme v poli *polePredku*. Tedy tak, že se podívám na index *krizovatka* (cílová křižovatka) a obsah této položky poslouží, jako další index, jehož obsah sledujeme. Tato činnost skončí tehdy, když je obsah indexu roven hodnotě *null*, a tedy je nalezen startovní vrchol. Po celou dobu rekurze ukládáme hodnoty indexů do seznamu *cesta*. Nevýhodou této rekonstrukce je fakt, že jsou křižovatky v objektu cesta poskládány opačně, ale jde to dobře řešit.

Metoda *NaplnListView* přijímá objekt třídy *ListView*. Instance této třídy vypadá takto:

Křižovatka	Délka
Žižkovo Náměstí_Jiskrova	0
Na Okruhu_Jiskrova	95
Na Okruhu_Jaselská	47
Na Okruhu_Terronská	65
Na Okruhu_Generála Píky	137
Žižkovo Náměstí_Generála Píky	105
Celková vzdálenost	449 metrů

Obrázek 18 - Komponenta třídy *ListView*

Vidíme, že komponenta *ListView* vytváří vlastně tabulku. Při nastavování jeho vlastností je třeba přidat daný počet sloupců. Obsah řádků se tvoří a naplňuje do *Listview* přímo v aplikaci právě pomocí metody *NaplnListView*. Na obrázku 18 je ukázáno, že tato komponenta třídy *ListView* je použita jako výstup programu. Komponenta má definovány dva sloupce. Sloupec *Křižovatka* zobrazuje posloupnost křižovatek tvořících nejkratší cestu. Druhý sloupec *Délka* vypisuje vzdálenost mezi křižovatkou na daném řádku a předchozí.

Tato komponenta je známá např. ze systému Windows. Tam se používá jako základní panel, na kterém se zobrazuje obsah složek. Komponenta *Listview* má několik režimů, jak zobrazovat své prvky. Jde např. o režim velké ikony, malé ikony, seznam, detaily. Zde na obrázku 18 stejně jako ve vlastním programu je použit mód detaily.

Metoda *NaplnListView* obsahuje *for* cyklus, který jde přes všechny prvky seznamu *cesta*. V každém průchodu *for* cyklem se vytvoří objekt *lvi* třídy *ListViewItem*. Do instance *lvi* pak vložíme název aktuální křižovatky a délku úseku mezi aktuální a předchozí křižovatkou. Pokud vytváříme první objekt třídy *ListViewItem*, je třeba za délku úseku mezi křižovatkami vložit nulu, protože první křižovatka nemá žádnou předchozí křižovatku. Pokud nevytváříme první objekt třídy *ListViewItem*, pak musíme odečíst vzdálenosti aktuální a předcházející křižovatky od startovního vrcholu. Ke zjištění vzdáleností daných křižovatek od startovního vrcholu použijeme pole *poleVzdalenosti*, kde jsou tyto informace uloženy. Toto pole je třeba oindexovat celočíselnou hodnotou, proto je možné použít slovník *KrizovatkyKeyIsString* vracející hodnotu *int*. Slovník je nutno oindexovat hodnotou *string*. Tu získáme ze seznamu *cesta*, ze kterého vybereme aktuální a předcházející křižovatku. Pro ujasnění postupu při tomto výpočtu je zde ukázka ze zdrojového kódu:

```
poleVzdalenosti[graf.KrizovatkyKeyIsString[cesta[i]]]-
poleVzdalenosti[graf.KrizovatkyKeyIsString[cesta[i - 1]]])
```

Pomocí prvního řádku kódu je zjištěna vzdálenost mezi aktuální křižovatkou a startovním bodem. V druhém řádku je zjištěna vzdálenost mezi předchozí křižovatkou a startovním bodem. Pokud tyto vzdálenosti odečteme, získáme délku úseku mezi oběma křižovatkami. Když máme vytvořen takto objekt třídy *ListViewItem*, vložíme ho do instance třídy *ListView* pomocí metody *Add*. Po ukončení *for* cyklu je vytvořena další položka do komponenty *ListView*. Tou je objekt *lviKonec*, kam se uloží celková délka nejkratší cesty. Celkovou délku cesty máme uloženou v poli *poleVzdalenosti* na indexu cílového vrcholu. Cílovou křižovatku máme uloženou jako typ *string*, je tedy třeba převést tento typ *string* na *int*. K tomu zase

použijeme slovník *KrizovatkyKeyIsString*. Tímto je objekt třídy *ListView* naplněn potřebnými informacemi o vyhledané nejkratší cestě.

Další metodou je metoda *Vysledek*, která přijímá objekt třídy *ListView*. Tato metoda slouží ke zpracování výsledku Dijkstrova algoritmu. Výsledná nejkratší cesta je uložena v poli *polePredku*, jehož obsah je třeba interpretovat tak, aby bylo možné ho jednoduše vypsat do komponenty *ListView*. V těle metody nejprve vytvoříme nový objekt typu *List<string>* a jeho referenci uložíme do instanční proměnné *cesta*. Dále vymažeme dosavadní obsah objektu třídy *ListView* pomocí metody *Clear*. Následně spustíme metodu *RekonstrukceCesty* a jako argument do ní vložíme cílovou křižovatku uloženou v proměnné *cil*. Tato metoda do seznamu *cesta* uloží křižovatky tvořící nejkratší cestu. Vzhledem k tomu, že jsou křižovatky v seznamu *cesta* uloženy opačně tedy od cíle ke startu, je třeba pořadí otočit. To provedeme metodou *Reverse*, která je interním členem třídy *List*. Nakonec naplníme objekt *ListView* tím, že spustíme metodu *NaplnListView*. Tu ovšem spustíme jen v případě, že počet prvků seznamu *cesta* je vyšší než jedna. Nejkratší cesta totiž musí obsahovat minimálně dvě křižovatky (startovní a cílovou). Pokud tomu tak není, znamenalo by to, že nejkratší cesta za zadaných podmínek neexistuje. V takovém případě je třeba tuto skutečnost oznámit uživateli pomocí okna *MessageBox*.

Metoda *IsIgnorovanaKrizovatka* vrací hodnotu typu *bool* a přijímá objekt *seznamIgnorovanychKrizovatek* typu *List<string>* a proměnnou *krizovatka* typu *string*. Metoda má za úkol zjistit, jestli křižovatka *krizovatka* se shoduje s některou z křižovatek v seznamu *seznamIgnorovanychKrizovatek* a zda tedy jde o ignorovanou křižovatku. Význam ignorovaných křižovatek je ten, že si uživatel může zvolit jejich libovolné množství a algoritmus nebude vyhledávat nejkratší cestu přes tyto křižovatky. V těle metody nejdříve deklarujeme proměnnou *vysledek* typu *bool* a inicializujeme ji na hodnotu *false*. Poté pokud reference na seznam *seznamIgnorovanychKrizovatek* neobsahuje hodnotu *null*, pak můžeme spustit cyklus *foreach*. Cyklus jde přes všechny křižovatky v seznamu *seznamIgnorovanychKrizovatek*. Každou křižovatku v seznamu porovnáme s křižovatkou v proměnné *krizovatka*. Pokud dojde v některém průchodu cyklu *foreach* k rovnosti těchto křižovatek, vložíme do proměnné *vysledek* hodnotu *true*. po skončení cyklu vrátíme proměnnou *vysledek* příkazem *return*. Pokud by reference na seznam *seznamIgnorovanychKrizovatek* obsahovala hodnotu *null*, znamená to, že žádné ignorované křižovatky nejsou zadány. V tom případě bychom rovnou vrátili proměnnou *vysledek*, která by měla hodnotu *false*.

Metoda *FiltrujSousedniVrchol* vrací hodnotu typu *bool* a přijímá tyto objekty těchto typů:

1. `int` – celočíselná hodnota znamenající index sousední křižovatky dané křižovatky v seznamu *seznamSousedu*. Proměnná se jmenuje *i*.
2. `Graf` – reference jménem *graf* na graf, který představuje datovou representaci městské sítě.
3. `TypCesty[]` – pole ukládající aktuální nastavení kategorií pozemních komunikací, přes které je možné vyhledávat. Toto pole se jmenuje *typCesty*.
4. `CheckBox` – objekt třídy *CheckBox* definuje, jestli uživatel vybral možnost dopravy „pěšky“ nebo „autem“. Pokud je tato komponenta zaškrtnutá, je vybrán způsob dopravy „pěšky“, v opačném případě je vybrán způsob „autem“. Objekt se jmenuje *chbPesky*.
5. `List<string>` – objekt představuje seznam, který ukládá uživatelem vybrané ignorované křižovatky. Reference na tento seznam se jmenuje *seznamIgnorovanychKrizovatek*.

Metoda umožňuje rozhodnut, jestli je možno danou sousední křižovatku na základě jejích vlastností použít pro výpočet nejkratší cesty. Vlastnostmi křižovatky je myšleno kategorie pozemních komunikací, způsob dopravy (autem nebo pěšky) a to, zda křižovatka není ignorovaná.

V těle metody nejprve deklarujeme proměnnou *vysledek* typu *bool* a inicializujeme ji na hodnotu *false*. Na začátku algoritmu této metody se kontroluje, jestli právě zkoumaný sousední vrchol není ignorovanou křižovatkou. To zjistíme zavoláním metody *IsIgnorovanaKrizovatka*. Dále je třeba určit, zda sousední vrchol spadá do některé z obojetných komunikací. To, které z obojetných komunikací jsou povoleny, udává pole *typCesty*. Pokud tedy do některé z takových kategorií sousední vrchol spadá, nastavíme proměnnou *vysledek* na hodnotu *true*. Pokud nespadá do žádné obojetné komunikace, následuje příkaz *switch*, který přijímá typ cesty aktuálního souseda. Příkaz switch obsahuje tyto tři větve:

1. typ cesty je *chodci* – pokud je checkbox udávající, jestli se pohybujeme pěšky zaškrtnut, nastavíme proměnnou *vysledek* na hodnotu *true*.

2. typ cesty je *silnice1AutaChodci* – jde o obojetný typ silnice 1. třídy. Proto pokud máme nastavený typ *silnice1AutaChodci* anebo *silnice1Auta*, tak v obou případech nastavíme proměnnou *vysledek* na hodnotu *true*.
3. typ cesty je *silnice1Auta* – v tomto případě je třeba zjistit, jestli je povolena silnice 1. třídy, která je jen pro auta. V případě, že ano, nastavíme proměnnou *vysledek* na hodnotu *true*.

Jestli sousednímu vrcholu nevyhovovala žádná větev tohoto algoritmu, hodnota proměnné *vysledek* se nezměnila a zůstává tedy stále *false*. Nakonec vrátíme obsah proměnné *vysledek* příkazem *return*.

Poslední metodou třídy je metoda *Start*, která už obsahuje vlastní Dijkstrův algoritmus. Do metody vstupují argumenty těchto typů:

1. `ListView` – komponenta, která slouží k zobrazení výstupu Dijkstrova algoritmu. Tedy k výpisu nejkratší cesty. Objekt se jmenuje *listView1*.
2. `TypCesty[]` – pole ukládající aktuální nastavení kategorií pozemních komunikací, přes které je možné vyhledávat. Toto pole se jmenuje *typCesty*.
3. `CheckBox` – objekt třídy *CheckBox* definuje, jestli uživatel vybral možnost dopravy „pěšky“ nebo „autem“. Pokud je tato komponenta zaškrtnutá, je vybrán způsob dopravy „pěšky“, v opačném případě je vybrán způsob „autem“. Objekt se jmenuje *chbPesky*.
4. `List<string>` – objekt představuje seznam, který ukládá uživatelem vybrané ignorované křižovatky. Reference na tento seznam se jmenuje *seznamIgnorovanychKrizovatek*.

Celé tělo metody je obaleno cyklem *do-while*, který končí tehdy, když je délka prioritní fronty rovna hodnotě nula. Nejprve si z instanční proměnné *fronta* odebereme první křižovatku metodou *Dequeue*. Tuto křižovatku si uložíme do proměnné *u* typu *string*. Křižovatka *u* představuje aktuální křižovatku, kterou Dijkstrův algoritmus řeší. Následně deklarujeme a inicializujeme seznamy *sousediU* a *vzdalenostiSouseduU*. Seznam *sousediU* je typu *List<string>* a *vzdalenostiSouseduU* je typu *List<double>*. Do seznamu *sousediU* budeme ukládat sousední křižovatky křižovatky *u* a do seznamu *vzdalenostiSouseduU* vzdálenosti od křižovatky *u* k jejím sousedům.

Dále musíme naplnit tato dvě pole vyhovujícími sousedy. Pro plnění polí sousedními křižovatkami slouží *for* cyklus, který musíme omezit tak, aby prošel jen sousední

vrcholy křižovatky *u*, které jsou uloženy v seznamu *seznamSousedu*. Informace o tom, na jakém indexu začínají sousední vrcholy křižovatky *u* v seznamu *seznamSousedu* zjistíme v seznamu *seznamKrizovatky* na indexu *u*. Abychom mohli indexovat seznam křižovatkou *u*, musíme změnit její interpretaci z řetězce na celé číslo, tedy z typu *string* na typ *int*. Ke změně této interpretace použijeme slovník *KrizovatkyKeyIsString*. Výslednou hodnotou uloženou ve slovníku už je možno indexovat seznam *seznamKrizovatek*. Náš *for* cyklus omezíme od čísla, které je uloženo na *u*-tém indexu do čísla na *u+1*-tém indexu v seznamu *seznamKrizovatek*. Na těchto indexech najdeme v seznamech *seznamSousedu* a *seznamDelek* samotné sousední vrcholy i vzdálenosti k nim od křižovatky *u*. Tato data ukládáme do seznamů *vzdalenostiSouseduU* a *sousediU*. Ke zjištění, jestli má daný sousední vrchol vyhovující vlastnosti (vede k němu uživatelem vybraná kategorie pozemní komunikace a nejde o ignorovanou křižovatku) a může být přidán do seznamu sousedů vrcholů *u*, slouží metoda *FiltrujSousedniVrchol*. Pokud tato metoda vrátí hodnotu *true*, vrchol bude přidán a v opačném případě přidán nebude.

Po ukončení přidávání sousedních křižovatek následuje další důležitá část Dijkstrova algoritmu. Tou je výpočet nových vzdáleností sousedů a jejich porovnání s dosavadními vzdálenostmi. Tato část je opět uzavřena ve *for* cyklu, který prochází všechny sousední vrcholu křižovatky *u*. Deklarujeme proměnnou *vzdalenost* typu *double* a inicializujeme ji na hodnotu, která je dána součtem vzdálenosti vrcholu *u* od startovního vrcholu a délky hrany (pozemní komunikace) k sousednímu vrcholu. Vzdálenost vrcholu u od startovního vrcholu máme uloženou v poli *poleVzdálenosti* na indexu *u*. Opět je nutno použít slovník *KrizovatkyKeyIsString* ke změně representace vrcholu mezi typem *string* a typem *int*. Délku hrany najdeme v seznamu *vzdalenostiSouseduU*. Následuje podmínka, kde porovnáváme novou vzdálenost sousedního vrcholu v proměnné *vzdalenost* s dosavadní vzdáleností (tuto dosavadní vzdálenost zjistíme v poli *poleVzdalenosti* na indexu daného sousedního vrcholu). Pokud je nová vzdálenost větší nebo rovna dosavadní vzdálenosti, neděláme nic. V opačném případě uložím nově vypočítanou vzdálenost do pole *poleVzdalenosti* na index aktuálního sousedního vrcholu. Na stejný index do pole *polePredku* uložíme křižovatku *u*. Jako poslední příkaz v této podmínce spustíme metodu *DecreaseKey* objektu *fronta*. Do této metody vložíme jako argumenty vzdálenost k sousednímu vrcholu tedy proměnnou *vzdalenost* a daný sousední vrchol. Potom, co bude ukončen cyklus *do-while* a tím pádem i celý Dijkstrův algoritmus, bude spuštěna metoda *Vysledek*. Do této metody vložíme objekt třídy *ListView*, který rovněž přijímá metoda *Start*.

10.7 Třída Form1

Třída *Form1* představuje hlavní formulář aplikace.

10.7.1 Instanční proměnné třídy

1. Třída obsahuje deklaraci objektu graf třídy Graf. Tato instance představuje samotný uliční model města.
2. Dále obsahuje objekt nazvaný *dijkstruvAlg*, který je instancí třídy *DijkstruvAlgoritmus*. Tento objekt implementuje funkčnost pro Dijkstrův algoritmus, který vyhledává nejkratší cestu v grafu.
3. pole typu *TypCesty[]* nazvané *typCesty*. Toto pole má velikost šest a slouží k uložení informací, přes které typy kategorií pozemních komunikací je možné vyhledávat nejkratší cestu. Povolené kategorie pozemních komunikací mají v poli uvedeny odpovídající hodnoty z výčtového typu *TypCesty*. Pokud je povoleno méně než všech šest typů pozemních komunikací, je na zbytku položek pole uložena hodnota *nic*.
4. seznam *seznamIgnorovanychKrizovatek* je typu *List<string>* a slouží k uložení libovolného počtu ignorovaných křižovatek. Tedy těch křižovatek, přes které algoritmus nevyhledává nejkratší cestu.

10.7.2 Metody třídy

Konstruktor třídy nejprve spustí metodu *SestrojGraf* objektu *graf* třídy *Graf*. Dále inicializuje objekt *dijkstruvAlg* třídy *DijkstruvAlgoritmus* a do jeho konstruktoru vloží referenci na instanci *graf*. Dále nastavíme tlačítku *buttonZadatNepovoleneKrizovatky* vlastnost *Enabled* na hodnotu *false*. Tato vlastnost zajišťuje to, jestli tlačítko bude mít odezvu vůči uživateli. Laicky řečeno komponenta s takto nastavenou vlastností *Enabled* „zešedne“ a není možné ji ovládat. Nakonec nastavíme vlastnost *Text* tlačítka *buttonZadatNepovoleneKrizovatky* na hodnotu *"Zadat ignorované křižovatky"*.

Třída obsahuje veřejnou vlastnost *SeznamIgnorovanychKrizovatek*, která je pouze pro zápis. Vlastnost je typu *string* a umožňuje přidávat nové položky do seznamu *seznamIgnorovanychKrizovatek*. Přidávání položek do seznamu funguje pouze tehdy, když do

této vlastnosti vkládáme objekty třídy *string*, které se nerovnají hodnotě *null*. Pokud by se vkládaný objekt rovnal hodnotě *null*, způsobí to vložení hodnoty *null* do samotného seznamu *seznamIgnorovanychKrizovatek* a tím jeho smazání. Smazání by bylo provedeno pomocí automatické správy jazyka C#.

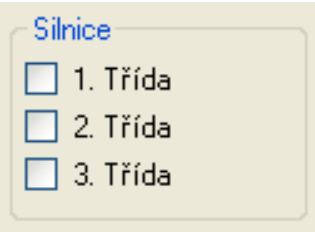
Metoda *numericUpDown1_ValueChanged* je obslužnou metodou události *ValueChanged* objektu *numericUpDown1* třídy *NumericUpDown*. Tato metoda umožňuje měnit nastavení vlastnosti *Enabled* tlačítka *buttonZadatNepovoleneKrizovatky*. Pokud je vlastnost *Value* objektu *numericUpDown* větší než nula a zároveň je referenční proměnná *seznamIgnorovanychKrizovatek* rovna hodnotě *null*, vlastnost *Enabled* tlačítka se nastaví na hodnotu *true*. V opačném případě se tato vlastnost tlačítka nastaví na hodnotu *false*. Uživatelem zadaná hodnota ve vlastnosti *Value* objektu *numericUpDown1* má význam počtu ignorovaných křižovatek.

Metoda *buttonZadatNepovoleneKrizovatky_Click* je obslužnou metodou události *Click* tlačítka *buttonZadatNepovoleneKrizovatky*. Metoda řeší naplnění seznamu *seznamIgnorovanychKrizovatek* křižovatkami, které bude program ignorovat. V těle metody si inicializujeme seznam *seznamIgnorovanychKrizovatek* novým objektem typu *List<string>*. Dále je zde *for* cyklus, který má svou řídící proměnnou *i* typu *int* nastavenou od hodnoty jedna a cyklus se ukončí, až přestane platit, že proměnná *i* je menší nebo rovna počtu ignorovaných křižovatek. Po každém průchodu cyklem se hodnota proměnné *i* inkrementuje. V těle *for* cyklu je vytvořen objekt *formZadaniKrizovatek* třídy *FormZadaniKrizovatek.* Do konstruktoru tohoto objektu vložíme řídící proměnnou *for* cyklu *i*, referenci na objekt *graf* a referenci na objekt třídy *Form1*. Tento objekt představuje formulář pro zadání zakázaných křižovatek. Uživatel v něm má možnost vybrat jakoukoli z křižovatek městské sítě, přes kterou algoritmus nebude vyhledávat nejkratší cestu. Tento formulář spustíme metodou *ShowDialog*. Parametry předané do konstruktoru objektu *formZadaniKrizovatek* nám umožní přijmout data zpět do objektu třídy *Form1*. Pokud je v některém průchodu *for* cyklu po uzavření formuláře *formZadaniKrizovatek* reference na seznam *seznamIgnorovanychKrizovatek* rovna hodnotě *null*, ukončíme *for* cyklus příkazem *break*. Po ukončení *for* cyklu se provede vložení hodnoty nula do vlastnosti *Value* objektu *numericUpDown1*. Nakonec zde máme podmínku, která sleduje, jestli reference na seznam *seznamIgnorovanychKrizovatek* je rovna hodnotě *null*. Pokud je v této referenční proměnné uložena hodnota *null*, vložíme do vlastnosti *Text* tlačítka *buttonZadatNepovoleneKrizovatky* řetězec *"Zadat ignorované křižovatky"*. V opačném případě do vlastnosti *Text* tohoto tlačítka vložíme řetězec *"Ignorované křižovatky zadány"*.

Metoda *comboBox_Click* je obslužnou metodou události *Click* obou objektů třídy *ComboBox*. Jde o objekty *comboBoxStart* a *comboBoxCil*, do kterých se zadává startovní a cílová křižovatka, mezi kterými bude Dijkstrův algoritmus hledat nejkratší cestu. V těle metody nejprve deklarujeme proměnnou typu *ComboBox* a vložíme do ní objekt *sender* přetypovaný z typu *object* na typ *ComboBox*. Objekt *sender* vstupuje do každé obslužné metody jakékoliv události a obsahuje referenci na instanci, která tuto událost vyvolala. Poté spustíme metodu *ComboBoxClick* statické třídy *MetodyProCombobox*. Do této metody vložíme referenci na objekt třídy *ComboBox*, který tuto událost vyvolal a ještě objekt třídy *Graf*.

Metoda *comboBox_TextChanged* je obslužnou metodou události *TextChanged* obou objektů třídy *ComboBox*, které jsou vytvořeny v této třídě. V těle metody nejprve deklarujeme proměnnou typu *ComboBox* a vložíme do ní objekt *sender* přetypovaný z typu *object* na typ *ComboBox*. Poté spustíme metodu *ComboBoxTextChanged* statické třídy *MetodyProCombobox*. Do této metody vložíme referenci na objekt třídy *ComboBox*, který tuto událost vyvolal a ještě objekt třídy *Graf*.

Metoda *ProjdiCheckBoxy* přijímá referenci na objekt třídy *GroupBox* a proměnnou *meze* typu *int*. Instance třídy *GroupBox* je kontejner, jde tedy o komponentu, na kterou je možno vkládat jiné komponenty (tlačítka, checkboxy, radiobuttony…). Tato komponenta vypadá takto:



Obrázek 19 - Ukázka komponenty GroupBox

Ukázka komponenty je použita z našeho programu. Vidíme, že tato komponenta obsahuje tři objekty třídy *CheckBox*. Instance třídy *GroupBox* je tedy kontejner. Proměnná meze inicializuje řídící proměnnou ve for cyklu.

V těle metody *ProjdiCheckBoxy* si deklarujeme pole s názvem *pole* typu *string*, do kterého uložíme názvy všech hodnot výčtového typu *TypCesty*. To uděláme pomocí metody *GetNames* třídy *Enum*. Metoda *GetNames* přijímá daný výčtový typ pomocí klíčového slova *typeof* a vrací pole typu *string*. Poté si definujeme proměnnou *enumerator* typu *IEnumerator*.

Tato proměnná umožňuje iterovat přes kolekce (např. objekt třídy *List*). Proměnnou *enumerator* inicializujeme pomocí metody *GetEnumerator* objektu třídy *GroupBox*, který nám vstoupil do metody. Metoda *GetEnumerator* vrátí kolekci komponent v přijímané instanci třídy *GroupBox*. My poté pomocí metod a vlastností proměnné *enumerator* budeme iterovat a pracovat se všemi komponentami v objektu třídy *GroupBox*. Následuje *for* cyklus, jehož řídící proměnná *i* je inicializována na hodnotu proměnné *meze*. *For* cyklus skončí tehdy, až řídící proměnná dosáhne hodnoty součtu proměnné *meze* a počtu komponent vstupního objektu třídy *GroupBox*. Počet komponent tohoto vstupního objektu zjistíme pomocí vlastnosti *Count*, kterou vyvoláme pomocí vlastnosti *Controls*. Vlastnost *Controls* totiž vrací kolekci komponent obsažených v kontejneru a tato kolekce nabízí vlastnost *Count*, která vrací počet prvků v kolekci.

V těle *for* cyklu metody *ProjdiCheckBoxy* použijeme metodu *MoveNext* proměnné *enumerator*, čímž se dostaneme na další prvek kolekce. Pokud tuto metodu spouštíme poprvé, tak by se mohlo zdát, že metoda *MoveNext* vrátí druhý prvek kolekce, protože objekt *enumerator* byl nastaven na první prvek kolekce. To ale není pravda, protože objekt *enumerator* při jeho inicializaci ukazuje před začátek kolekce. Proto tedy po prvním zavolání metody *MoveNext* objekt *enumerator* ukazuje na první prvek dané kolekce. Dále použijeme vlastnost *Current* objektu *enumerator*, která vrátí aktuální prvek kolekce, na který *enumerator* ukazuje. Vrácený prvek je typu *object*, takže ho přetypujeme na typ *CheckBox* a uložíme do referenční proměnné *chb* stejného typu. Pokud vlastnost *Name* tedy jméno aktuálního checkboxu je „silnice1“ a zároveň je tento *checkBox* zaškrtnutý vložíme do pole *typCesty* na index *i* hodnotu *silnice1Auta* z výčtu *TypCesty*. Pokud tento checkbox zaškrtnutý není, tak na index *i* do pole *typCesty* vložíme hodnotu *nic* z výčtu *TypCesty*. Jestli je instance třídy *CheckBox* zaškrtnutá nebo není, zjistíme podle hodnoty vlastnosti *Checked*. Když tato booleovská vlastnost nabývá hodnoty *true*, je objekt třídy *CheckBox* zaškrtnutý, v opačném případě zaškrtnutý není. Pokud se aktuální checkBox nejmenuje „silnice1“ děje se toto:

Zjišťujeme, jestli je náš objekt třídy *CheckBox* zaškrtnutý.

Pokud zaškrtnutý je, musíme zjistit jaká hodnota výčtového typu *TypCesty* odpovídá jeho názvu. To určíme porovnáváním vlastnosti *Name* checkboxu s prvky pole *pole*, v kterém jsou uloženy hodnoty výčtového typu *TypCesty*. Definujeme pomocnou proměnou *j* typu *int*, pomocí které budeme indexovat pole *pole*. Jednotlivé položky tohoto pole budeme porovnávat s názvem checkboxu uloženým ve vlastnosti *Name*. V případě nalezení rovnosti tuto proměnou *j* přetypujeme na typ *TypCesty* a uložíme na index *i* do pole *typCesty*. Takto přetypovaná proměnná *j* znamená přímo hodnotu daného výčtového typu. Dělat to tímto

způsobem je možná zvláštní, ale problém je v tom, že není možnost jak převést název aktuálního checkboxu na danou hodnotu výčtového typu. Přesto, že obě tyto hodnoty nesou stejnou informaci, je třeba to obcházet pomocí prohledávání pole s hodnotami výčtu a uložení přetypované hodnoty na typ daného výčtu.

Pokud zaškrtnutý není, tak se na index *i* pole *typCesty* uloží hodnota *nic* výčtového typu *TypCesty*.

Další metodou třídy *Form1* je metoda *NastavPoleTypuCest*. Tato metoda zajišťuje nastavení položek pole *typCesty* na základě nastavení obou objektů třídy *GroupBox*. Aplikace obsahuje *groupBoxSilnice* a *groupBoxMistniKomunikace*. Objekt *groupBoxSilnice* umožňuje nastavit, které třídy silnic jsou povoleny. Objekt *groupBoxMistniKomunikace* umožňuje nastavit, které třídy místních komunikací jsou povoleny. V těle metody se dvakrát spustí metoda *ProjdiCheckBoxy* ovšem pokaždé s jinými parametry. Poprvé do metody vstoupí reference na objekt *groupBoxSilnice* a celočíselná hodnota nula. Podruhé do metody vstoupí reference na objekt *groupBoxMistniKomunikace* a celočíselná hodnota tři. Vkládané celočíselné hodnoty udávají, kolik se prošlo objektů třídy *CheckBox* v předchozím případě. V prvním případě spuštění metody *ProjdiCheckBoxy* žádný předchozí případ nebyl a proto se zpracovalo celkem nula checkboxů a tak do metody vložíme číslo nula. Při druhém volání této metody se v předchozím případě zpracovaly tři instance třídy CheckBox, proto do metody vložíme hodnotu tři.

Metoda *checkBoxPesky_CheckedChanged* je obslužnou metodou události *CheckedChanged* objektu *checkBoxPesky* třídy *CheckBox*. Tento checkbox udává, jestli je vybrán typ dopravy „autem“ nebo „pěšky“. Pokud je instance *checkBoxPesky* zaškrtnutá, pak uživatel nastavil možnost dopravy „pěšky“, v opačném případě je vybrána možnost dopravy „autem“.

V těle metody je potřeba zjistit, jestli je objekt *checkBoxPesky* zaškrtnutý nebo není. V případě, že je, se pole naplní všemi hodnotami obojetných typů kategorií pozemních komunikací a hodnotou *silnice1AutaChodci* pro silnici 1. třídy. V případě, že tento checkbox zaškrtnut není, tak se spustí metoda *NastavPoleTypuCest*.

Metoda *checkBox_CheckedChanged* je obslužná metoda události *CheckedChanged* objektů třídy *CheckBox*, které jsou umístěny na obou kontejnerech třídy *GroupBox*. V těle této metody se spustí metoda *NastavPoleTypuCest* ale jen v případě, že objekt *checkBoxPesky* není zaškrtnut.

Metoda *NastavPoziceComboboxu* počítá vlastnosti *Location* a *Size* objektů *comboBoxStart* a *ComboBoxCil*. Pro objekt *label2* počítá pouze vlastnost Location. Metoda

tyto vlastnosti počítá na základě aktuální šířky hlavního okna aplikace. Metoda řeší to, aby byl stále stejný poměr velikostí a umístění comboboxů v okně při změně jeho velikosti.

Metoda *VratPocetZnakuNejdelsihoJmena* vrací počet znaků křižovatky, která má nejdelší název ze všech křižovatek, které jsou výsledkem vyhledávacího algoritmu. V těle metody je inicializována proměnná *delka* typu *int* na hodnotu *-1*. Postupně projdeme všechny křižovatky v komponentě listview. V případě, že nalezneme křižovatku s delším názvem než je hodnota proměnné *delka*, tak počet znaků jejího názvu vložíme do proměnné *delka*. Na konci metody vrátíme proměnnou *delka*, kde bude počet znaků nejdelšího názvu.

Metoda *VratTecky* vrací řetězec a přijímá dva parametry. První je *delkaNazvu* typu *int*, který obsahuje počet znaků aktuální křižovatky. Druhý je *delkaNejdelsihoNazvu*. Ten je také typu *int* a obsahuje počet znaků nejdelšího názvu křižovatky. Metoda se používá pro export výsledku aplikace do textového souboru. Z důvodu přehlednosti je potřeba, aby křižovatky tvořící nejkratší cestu a vzdálenosti mezi nimi tvořily dva sloupce. Tato metoda vrací řetězec teček, který má proměnlivou délku v závislosti na rozdílu mezi počty znaků v názvech křižovatek. Konkrétní počet teček je určen rozdílem mezi parametry *delkaNejdelsihoNazvu* a *delkaNazvu*. K tomuto výsledku je ještě přičtena konstanta *2*, která posune o dvě tečky sloupec vzdáleností, aby nebyl nalepený na názvu nejdelší křižovatky. Řetězec teček je tvořen pomocí *for* cyklu, ve kterém je proměnná *tecky* typu *string*. K obsahu této proměnné se připojují další tečky pomocí operátoru +.

Metoda *ZapisTypyCest* souvisí také se zápisem do textového souboru. Slouží k tomu, že do něho zapíše nastavení kategorií pozemních komunikací, přes které je vyhledána nejkratší cesta. K tomuto je využito pole *typCesty*, kde jsou uloženy povolené komunikace. Do souboru jsou vypsány ty položky, které se nerovnají hodnotě *nic*.

Metoda *buttonExport_Click* je obslužnou metodou tlačítka *buttonExport*, které umožňuje exportovat výslednou nejkratší cestu do textového souboru. Exportovat je možné pouze v případě, že máme vygenerovanou cestu v komponentě listview. Pokud tomu tak je, spustíme metodu *VratPocetZnakuNejdelsihoJmena*, kterou zjistíme počet znaků nejdelšího názvu křižovatky a výsledek uložíme do proměnné *pocetZnakuNejdelsihoJmena*. Vytvořenému objektu *sfd* třídy *SaveFileDialog* nastavíme vlastnost *Filter* na hodnotu *"txt | *.txt"*, čímž umožníme, aby bylo možné ukládat jen soubory s příponou txt. Dále tento ukládací dialog otevřeme, vybereme místo uložení a pokud okno zavřeme jinak než kliknutím na tlačítko OK, končíme. V opačném případě do souboru zapíšeme povolené kategorie místních komunikací pomocí metody *ZapisTypyCest*. Dále zrekonstruujeme postupně každý řádek z listview do proměnné *radek* typu *string*. To provedeme tak, že do proměnné *radek*

vložíme obsah buňky prvního sloupce, k němu připojíme potřebný počet teček, který prací metoda *VratTecky* a nakonec připojíme obsah buňky druhého sloupce. Tyto řádky zapíšeme postupně do souboru.

Poslední metodou třídy *Form1* je metoda *buttonVyhodnotit_Click*. Jde o obslužnou metodu události *Click* tlačítka *buttonVyhodnotit*. Metoda zajišťuje výpočet nejkratší cesty vzhledem k zadaným datům. Zadanými daty je myšleno startovní a cílová křižovatka, množina ignorovaných křižovatek a dané nastavení kategorií pozemních komunikací, přes které je možné vyhledávat.

V těle metody se deklarují proměnné *start* a *cil*, obě jsou typu *string* a ukládají se do nich názvy startovní a cílové křižovatky. Do proměnné *start* uložíme vlastnost *Text* objektu *comboBoxStart* třídy *ComboBox*. Do proměnné *cil* uložíme vlastnost *Text* objektu *comboBoxCil* třídy *ComboBox*. Dále je potřeba vymazat dosavadní obsah objektu *listView1* třídy *ListView* pomocí metody *Clear*. Následuje podmínka, která hodnotí, jestli křižovatky v proměnných *start* a *cil* existují. K tomuto rozhodnutí použijeme slovník *krizovatkyKeyIsString* a jeho metodu *ContainsKey*. Metoda *ContainsKey* přijímá v tomto případě typ *string* a vrací hodnotu *bool*, podle toho, jestli klíč existuje nebo neexistuje. V naší podmínce tedy použijeme tuto metodu celkem dvakrát a to pro startovní a cílovou křižovatku. Výsledky obou metod spojíme v jeden pomocí operátoru logického součinu (and). Pokud se má vykonat tělo podmínky, musí platit, že obě křižovatky existují. V případě, že tedy obě křižovatky existují, spustíme metody *Inicializace*, do které vložíme startovní a cílovou křižovatku a *Start*, kam vložíme objekty *listView1*, pole *typCesty*, checkbox *checkBoxPesky* a seznam *seznamIgnorovanychKrizovatek*. Na závěr metody je třeba provést několik příkazů, které již nespadají do žádné podmínky. Jde o vložení hodnoty *null* do objektu *seznamIgnorovanychKrizovatek*. Tím se postaráme o smazání tohoto objektu, protože na jeho oblast v paměti již neukazuje z aplikace žádná referenční proměnná. Dále do vlastnosti *Text* tlačítka *buttonZadatNepovoleneKrizovatky* vložíme řetězec *"Zadat ignorované křižovatky"*. A nakonec do vlastnosti *Value* objektu *numericUpDown1* vložíme hodnotu nula.